Ph.D. Thesis

Supporting the Grow-and-Prune Model for Evolving Software Product Lines

Leticia Montalvillo Mendizabal

June, 2018

eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea

Web Engineering Research Group

Supervisor: Prof. Dr. Oscar Díaz

Supporting the *Grow-and-Prune* Model for Evolving Software Product Lines

Dissertation
presented to
the Department of Computer Languages and Systems of
the University of the Basque Country
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy
("*international*" mention)

Leticia Montalvillo Mendizabal

Supervisor: *Prof. Dr. Oscar Díaz García*
San Sebastián, Spain, 2018

This work was hosted by the *University of the Basque Country* (Faculty of Computer Science). The author enjoyed a doctoral grant from the University of The Basque Country (UPV/EHU) from 2014 to 2018. The work was co-supported by the *Spanish Ministry of Education*, the *European Social Fund* and the *University of The Basque Country* (UPV/EHU) under contracts TIN2014-58131-R, TIN2011-23839 (Scripting), and OSADATU UFI11/19.

"A man can only attain knowledge with the help of those who possess it. This must be understood from the very beginning. One must learn from him who knows."

– *George Ivanovich Gurdjieff*.

Resumen

El contexto de esta tesis se enmarca dentro de la ingeniería de las Líneas de Producto Software (LPS), y de como soportar su evolución mediante el paradigma conocido como *grow-and-prune* (traducido del inglés como “crecimiento-y-poda”). Se ha seguido el enfoque *Design Science Research*, para identificar y solucionar los problemas que aparecen en el contexto de las SPLs que evolucionan mediante el paradigma grow-and-prune. A continuación se resumen el contexto de la tesis, los problemas que se abordan, y las contribuciones científicas que esta tesis aporta al área de las LPS.

Las Líneas de Producto Software (LPS).

En los últimos años, la ingeniería de LPS ha ganado un impulso considerable. Empresas como Boeing, Bosch, General Motors, Philips o Siemens recurren a las LPS para ampliar su abanico de productos software, aumentar el retorno de la inversión, acortar el tiempo de comercialización y mejorar la calidad del software. En esencia, las LPS tienen como objetivo respaldar el desarrollo de toda una familia de productos de software, a través de la reutilización sistemática de los llamados artefactos reutilizables. Con este fin, el desarrollo de las LPS se divide en dos procesos ínter-relacionados: (1) ingeniería de dominio (ID), que se encarga de definir cual es el alcance, cuales son las características (*features*) de la LPS y su variabilidad, y desarrolla los artefactos software reutilizables; y (2) ingeniería de aplicaciones (IA), donde los productos se construyen mediante la reutilización de aquellos artefactos reutilizables desarrollados en la ID, y resolviendo la variabilidad de los mismos.

Con el fin de obtener el máximo beneficio, la ingeniería de LPS tiene como objetivo automatizar la derivación del producto a partir de los artefactos reutilizables. Estos artefactos reutilizables se caracterizan en términos de características, es decir, aspectos distintivos visibles para el usuario, cualidades o características de los productos de la LPS. Idealmente, la derivación del producto se limita a indicar el conjunto de características que exhibirán los productos, sin la necesidad de ningún desarrollo extra en el producto obtenido. En este escenario, los productos se obtienen a través de una "Familia de Producto Altamente Configurable" (FPAC), donde casi el 100% del esfuerzo lo dedican los ingenieros de dominio a la construcción de artefactos reutilizables, y la personalización del producto no existe. Sin embargo, los escenarios reales están lejos de ser ideales.

Evolución en las LPS.

En primer lugar, llegar a una FPAC no es trivial, sino el resultado de un largo viaje que dura años. Es por eso, que en la práctica existen etapas intermedias, (por ejemplo, "Plataforma", "Línea de productos de software"), en las que el conjunto de artefactos reutilizables no cubre completamente las necesidades de los productos, y por lo tanto , los ingenieros de aplicación necesitan desarrollar las funcionalidades restantes ellos mismos. En la práctica, las LPS se inician frecuentemente con una versión parcial de los artefactos reutilizables, que se obtienen mediante la refactorización de (un subconjunto de) variantes de productos ya existentes, y que se amplían gradualmente con funcionalidades más nuevas extraídas de productos que se han derivado de ella. Durante este viaje, los activos centrales y los productos coexisten mientras el código del producto se refactoriza progresivamente y se migra al dominio de los artefactos reutilizables.

Segundo, incluso si la LPS ha llegado a un nivel de reutilización como el de las FPAC, las LPS pueden alcanzar una escala y complejidad que las hace inviables para evolucionar en un corto período de tiempo. Por lo tanto, cuando una organización tiene como objetivo reaccionar con rapidez a los eventos del mercado o las solicitudes urgentes de los clientes, se necesitan estrategias para respaldar la adaptación rápida, con extensiones y personalizaciones específicas del producto. La conclusión final es que la personalización del producto no siempre se puede evitar.

Modelo *grow-and-prune*.

Este modelo distingue entre dos etapas durante la evolución de las LPS: *crecimiento* (grow) y *poda* (prune). El modelo permite que los productos *crezcan* una vez derivados de la base los artefactos reutilizables de la ID. Durante la poda, aquellas funcionalidades (de producto) que se consideran útiles se promocionan a la ID (es decir, a artefactos reutilizables) mediante la refactorización y la fusión software (*merge*). En este modelo, tanto los artefactos reutilizables como los productos evolucionan conjuntamente a través de dos caminos: mientras el código de producto (IA) se porta progresivamente al dominio de los artefactos reutilizables (ID) central, siguiendo la ruta de *retroalimentación*, los productos también se pueden actualizar con nuevas funcionalidades y correcciones de fallas lanzadas por los ingenieros de dominio siguiendo la ruta de *actualización*.

En el contexto de la evolución de las LPS con el modelo *grow-and-prune*, surgen una serie de problemas. A continuación se listan los problemas abordados mediante esta tesis.

Problema 1: analizar de las personalizaciones de los productos.

Como hemos apuntado, el modelo *grow-and-prune* permite a los productos que implementes aquellas funcionalidades que no vienen soportadas por los artefactos reutilizables de la ID. En el contexto de esta tesis, a los cambios y desarrollos software llevados a cabo por los ingenieros de la aplicación se les llama *personalizaciones*. Una vez la etapa de crecimiento (grow) ha concluido, los ingenieros del dominio

han de empezar la poda (*prune*). Dicha poda requiere que los ingenieros del dominio tengan que analizar e inspeccionar todos los productos uno a uno, para determinar qué personalizaciones han desarrollado los productos, y cuales son adecuadas para formar parte del elenco de los artefactos reutilizables. Esto es clave, porque la nueva versión de los artefactos reutilizables ha de satisfacer los requisitos de los productos.

El problema principal es que identificar y analizar las personalizaciones de los productos es costoso y propenso a errores. Esto es debido a varias causas. Primero, las LPS cuentan con un gran número de artefactos, y productos. Es por ello, que al finalizar el proceso de *crecimiento de los productos,* los ingenieros cuentan con cientos de personalizaciones a analizar. Segundo, este análisis se lleva a cabo a un nivel de abstracción de bajo nivel, a nivel de archivo y de código fuente. Por el contrario, las LPS requieren que este análisis sea hecho a niveles más altos de abstracción, es decir, a nivel de *características* y *productos*. Los archivos son una noción de implementación. Por el contrario, "producto" y "característica" son conceptos abstractos que se desarrollan a través de los archivos. Es por eso, el análisis debe abstraerse de los archivos y reformularse en términos de "producto" y "característica". Finalmente, hay una falta de visualizaciones dedicadas para el análisis de las personalizaciones de las LPS. Las utilidades de *diff* tradicionales no escalan a los posibles miles de cambios que podrían estar involucrados en una LPS. Es por eso que se necesitan herramientas apropiadas, no solo para identificar las personalizaciones de los productos en los niveles de abstracción de *producto* y *característica*, sino también para visualizarlos "correctamente".

Si esta problemática no se aborda, los ingenieros del dominio pueden fallar a la hora de identificar correctamente las personalizaciones, y por consecuencia ofrecer una nueva versión de artefactos reutilizables que no satisfaga las necesidades de los productos. Esto, a su vez hace que los productos vuelvan a desarrollar sus propias personalizaciones, reutilizando cada vez menos los artefactos de la ID. En este escenario, la LPS se arriesga a una degradación de la reutilización, i.e. una situación en la que los productos han degenerado tanto de la LPS, en la que los productos ya no reutilizan artefactos de las LPS ya que ya no se consideran útiles; o si se consideran útiles, la integración en el producto es tan costosa que no compensa su reutilización (debido al alto número de conflictos entre los nuevos artefactos reutilizables y el código personalizado del producto). En este escenario, se produce una caída de la productividad de la IA, y, a su vez, aumentaría el tiempo de comercialización de los productos.

Con el objetivo de ayudar a los ingenieros de la IA a analizar las personalizaciones, esta tesis se plantea las siguientes preguntas de investigación.

- P1: ¿Cuáles son las necesidades de información para el análisis de personalización? Para cada necesidad de información, ¿cuánto tiempo se necesita para obtenerla?
- P2: ¿Hasta qué punto se pueden satisfacer las necesidades de información previas a través de un almacén de datos? Si es así, ¿cómo se vería su esquema de estrella?
- P3: ¿Cómo se puede visualizar el análisis de personalización?

Esta tesis contribuye a las anteriores preguntas de esta manera:

- P1. Se han identificado las necesidades de información que surgen durante los escenarios de evolución de LPS ("evolución de características" y "evolución del producto"). La importancia de cada necesidad de información y el tiempo requerido para obtenerlas se validan a través de un cuestionario entregado a los profesionales SPL.
- P2. Se ha desarrollado la herramienta *CustomDIFF*, con un enfoque de almacén de datos, para el análisis de personalización que usa Git como el sistema de control de versiones desde donde se obtienen los datos de hechos, y pure::variantes como framework de desarrollo de la LPS. El esquema de estrella diseñado permite a los ingenieros de LPS agregar hechos de personalización a lo largo de diferentes niveles de abstracción, tales como, producto, característica, activo central y componente. *CustomDIFF* ha sido probado con profesionales de las LPS para evaluar su utilidad y uso de uso para el análisis de personalización.
- P3. Recurrimos a diagramas de aluvión para visualizar el esfuerzo de personalización de un vistazo. Estos diagramas son un tipo de diagramas de flujo. Aquí, el flujo representa el esfuerzo de personalización que va desde las características a los productos de la LPS, si es que se ha necesitado personalización.

Problema 2: contrarrestar el problema de la fusión infernal (*merge hell*).

Siguiendo el modelo *grow-and-prune*, la etapa de la *poda* (prune) requiere que las personalizaciones de productos que se consideran útiles se integren en la base de los activos básicos mediante la fusión (*merge*) y refactorización. Tenga en cuenta que, debido a la fase previa de *crecimiento*, los productos podrían haber divergido sustancialmente entre sí. Durante la *poda*, los ingenieros deben conciliar estas divergencias resolviendo los conflictos que surgen de la combinación de personalizaciones dispares de productos. Cuando el tiempo para resolver estos conflictos excede el tiempo necesario para realizar los cambios originales, nos encontramos en la denominada situación de problema de fusión (fusión infernal).

El problema pues, es que la fusión y refactorización de las personalizaciones de productos es difícil y lleva mucho tiempo. Esto es causado por las divergencias que se llevan a cabo durante el proceso de *crecimiento*. Cuanto más altas sean las divergencias entre los productos, mayores serán los conflictos y la complejidad para resolverlos. Lo que se necesita es una forma para que los equipos de productos no se desvíen mucho los unos de los otros, y por lo tanto, facilitar la fusión posterior. Si no, esto causaría una bajada de la productividad de los ingenieros del dominio, que solo dispondrían del tiempo para *portar* estas personalizaciones, desatendiendo otras tareas como las de perfeccionar y desarrollar mejoras en las *características* ya existentes, o en nuevas solicitudes de características. Esto comprometería a su vez el tiempo de lanzamiento de la nueva versión de los artefactos reutilizables.

Con el objetivo de reducir el problema del merge, esta tesis se plantea las siguientes preguntas de investigación:

- P1: ¿Cómo se lleva a cabo el modelo *grow-and-prune* en la práctica?
- P2: ¿Cuáles son las características del problema de fusión en las LPS? Y, ¿cómo podemos disminuir el problema de fusión en las LPS?

Esta tesis contribuye a las anteriores preguntas de esta manera:

- P1. Descripción de los roles y las interacciones que se entremezclan en un enfoque de grow-and-prune, motivado por el caso de la empresa Danfoss Drives.
- P2. Caracterización del problema de fusión en las LPS y cómo difiere del problema que aparece en el desarrollo tradicional software. Proponemos una nueva práctica denominada *code peering*, que se lleva a cabo durante el proceso de *crecimiento*, mediante la cual los ingenieros del producto inspeccionan y comparan el código de otros productos con el suyo propio. Esto tiene la intención de promover la reutilización temprana entre los diferentes equipos de IA, con el objetivo de disminuir las divergencias entre ellos, y en consecuencia disminuir el problema de fusión posterior. Esto plantea la pregunta de si merece la pena desviar la atención de los desarrolladores del producto para facilitar la posterior eliminación por parte de los ingenieros de dominio. Usando la teoría de *Attention Investment*, presentamos cuatro principios de diseño que impulsan cómo se puede introducir el peering de código para el desarrollo de las LPS.
- P2. Realización de estos principios en el prototipo *PeeringHub*, una herramienta que soporta el *code peering* a través de: (1) una extensión de Chrome que mejora GitHub con barras de interconexión que brindan información breve sobre qué características están cambiando otros productos, (2) una web aplicación que proporciona visualizaciones de alto nivel basadas en aluviones que indican las características disponibles para el *code peering*, y (3) comparaciones tridireccionales (*3-way diff*) basadas en características para que los ingenieros de producto puedan analizar cómo un producto está cambiando el código de una característica, con respecto al suyo propio.

Problema 3: sincronizar artefactos reutilizables y productos

La etapa de *poda* del modelo *grow-and-prune*, requiere la propagación de cambios entre los artefactos reutilizables (ID) y los productos (IA), de modo que ambas partes estén sincronizadas. Esto introduce dos operaciones de sincronización: la propagación de *actualización* (desde la ID a la IA) y la propagación de la *retroalimentación* (desde la IA a la ID). Son los ingenieros de la LPS quien tiene que ejecutar dichas operaciones. Y aquí es esencial que se consiga una sincronización completa y correcta y entre artefactos reutilizables y productos.

El problema que surge en este contexto es que ejecutar la propagación de *actualización* como la propagación de retroalimentación es un proceso tedioso y propenso a errores. Esto es debido principalmente a dos causas. Primero, los sistemas de control de versiones (SCV) son sistemas que facilitan el desarrollo software, pero no están adecuados las peculiaridades de las LPS. Es decir, los SCV tradicionales no soportan de una manera nativa las operaciones de sincronización entre artefactos

reutilizables y productos. Segundo, no hay directrices claras de cómo el desarrollo software se ha de organizar en los SCV, mediante los modelos de ramas.

Esta problemática dificulta la sincronización entre los artefactos reutilizables y los productos. Si ambas partes, ID e IA, no están sincronizadas, los productos no recibirán las últimas correcciones de errores y nuevas funcionalidades disponibles Por lo tanto, los productos tendrían que desarrollarlos ellos mismos, lo que hace que los ingenieros de aplicaciones sean menos productivos e incrementan el tiempo de salida al mercado de dichos productos. Por otro lado, cuanto menos sincronizados estén los productos con los artefactos reutilizables, más se desviarán los productos. Esto significa que los productos querrán reutilizar los artefactos reutilizables con menor frecuencia, ya que las actualizaciones requerirían un esfuerzo de integración cada vez mayor. Cuanto más larga sea la demora de actualización, mayor será la cantidad de actualizaciones disponibles recientemente, y mayores serán los conflictos entre los desarrollos de ID e IA, ocasionando finalmente un degradación de la reutilización.

Con el objetivo de ayudar a la sincronización de los artefactos reutilizables y productos, esta tesis se plantea las siguientes preguntas de investigación:

- P1: ¿Cómo se pueden organizar los productos y los activos centrales en SCV (por ejemplo, Git) y cómo sería el modelo de ramas?
- P2: ¿Cómo pueden los SCV web (por ejemplo, Github) ayudar a la sincronización de los artefactos reutilizables y los productos?

Esta tesis contribuye a las anteriores preguntas de esta manera:

- P1. Proponemos una arquitectura de repositorio SCV, que distingue entre el repositorio *artefacto*s, donde tiene lugar la ingeniería de dominio, y los repositorios de *productos*, donde se produce la ingeniería de producto. Además, ofrecemos modelos de ramas para cada repositorio en el que operan las acciones de sincronización.
- P2. Desarrollamos la semántica operacional para las acciones de sincronización. El modelo de ramas anterior permite expresar las operaciones de sincronización en términos de construcciones SCV básicas (*branch*, *commit*, *merge*). Esto a su vez implica que los desajustes eventuales que se producen durante la sincronización se resuelven à la SCV, es decir, que resaltan la diferencia entre distintas versiones del mismo artefacto (tradicionalmente, utilizando la opción diff en SCV). Por lo tanto, no apuntamos a la sincronización automática. Nuestro objetivo es mucho más humilde: aprovechar los populares mecanismos de SCV para que los ingenieros de la LPS logren la sincronización de una manera similar a lo que hacen para los productos individuales. Sin embargo, esto da como resultado una brecha conceptual entre cómo se conciben las rutas de sincronización y cómo se realizan hasta la fusión completa. Para cerrar esta brecha, proponemos extender los SCV con las operaciones de sincronización SPL.
- P2. Como una prueba de concepto, desarrollamos *GitLine*, una extensión de navegador para Firefox, que amplía GitHub con operaciones de sincronización

LPS. Con un solo clic, los ingenieros de producto ahora pueden (1) generar repositorios de productos desde un repositorio de artefactos, a lo largo de una determinada configuración de características, (2) ejecutar propagaciones de *actualización* para actualizar el producto con nuevas versiones de artefactos reutilizables y (3) ejecutar propagación de *retroalimentación* de aquellas personalizaciones de productos que se quieran migrar a la ID.

Summary

Software Product Line (SPL) engineering has gained considerable momentum. Top leading companies such as Boeing, Bosch, General Motors, Philips or Siemens resort to SPLs to broaden their software portfolio, increase return on investment, shorten time to market, and improve software quality. Full benefits of SPLs are achieved through automating product derivation out of reusable core-assets. Ideally, product derivation is limited to indicating the set of features to be exhibited by products, with no further need for product development. However, achieving such a degree of reuse is not a one-shot effort but a many year-long journey. Hence, companies often rely on intermediary stages in which product teams need to change the core-assets as part of the product derivation process. In this context, both core-assets and products need to co-evolve.

The so-called *grow-and-prune* model has proven great flexibility to incrementally evolve an SPL by letting the products *grow,* and later *prune* the product functionalities deemed useful by refactoring and merging them back to the SPL core-asset base. Herein, both core-assets and products co-evolve by means of two sync paths: while product code is progressively ported to the core-asset realm following the *feedback path*, products are upgraded with newer functionalities and bug-fixes released by domain engineers following the *update path*.

On this ground, this Thesis aims at supporting the *grow-and-prune* model as for *initiating* and *enacting* the pruning. *Initiating* the pruning requires SPL engineers to conduct customization analysis, i.e. analyzing how products have changed the core-assets. Customization analysis aims at identifying interesting product customizations to be ported to the core-asset base. However, existing tools do not fulfill engineers needs to conduct this practice. To address this issue, this Thesis elaborates on the SPL engineers' needs when conducting customization analysis, and proposes a data-warehouse approach to help SPL engineers on the analysis.

Once the interesting customizations have been identified, the pruning needs to be *enacted*, by merging and refactoring product customizations into the core-asset base. However, this might cause a merge hell, in cases where there is a large number of conflicts when disparate product developments need to be reconciled. To address this issue, this Thesis proposes code peering, i.e. a practice whereby product engineers inspect and compare other products' code with their own code. This is intended to promote early reuse across product teams with the aim of lessening the merge problem. We discuss four design principles that drive how code peering can be introduced for SPL development, and realize them through a prototype. Eventually, product code needs to be ported to the core-asset realm, while products are upgraded with

newer functionalities and bug-fixes available in newer core-asset releases. Herein, synchronizing both parties through sync paths is required. However, the state of-the-art tools are not tailored to SPL sync paths, and this hinders synchronizing core-assets and products. To address this issue, this Thesis proposes to leverage existing Version Control Systems (i.e. *git*/*Github*) to provide sync operations as first-class constructs.

Acknowledgements

Let me start with a quote:

> *"It is good to have an end to journey toward; but it is the journey that matters, in the end."*
>
> *–Ursula K. Le Guin*

A PhD resembles a journey, at least to me. In that sense, what matters is not the achievement of a doctoral degree (the end), but the skills I sharpened, the knowledge I gained, and the experiences I lived throughout it (the journey). I owe gratitude to all the people who travelled with me during this journey/rollercoaster.

First, I owe my deepest gratitude to my supervisor Professor Oscar Díaz. There are many things I learnt from him. I am specially grateful for his socratic way of teaching, for inculcating us the importance of the problem statement, for his patience, his continuous encouragement, his wise council, and for always seeking the excellence.

I would like to show my gratitude to my teammates too, who turn the lab into a nice and stimulating working environment, for encouraging me when we were on a deadline, for bringing cookies and cakes to the coffee breaks, and for putting up with me whenever I started to talk about product lines, and "my things". It has been a pleasure to have all of you as teammates: Jon Iturrioz, Arantza Irastorza, Maider Azanza, Felipe Ibañez, Iker Azpeitia, Cristóbal Arellano, Gorka Puente, Josune de Sosa, Jokin García, Itziar Otaduy, Iñigo Aldalur, Juanan Pereira, Jeremías Pérez, and Haritz Medina.

I am indebted to Thomas Fogdal, functional manager at Danfoss Drives company, for giving me the opportunity to make a research visit at their place. I am grateful for his willingness to collaborate, his open mind, his generous spirit, and his Christmas events, where he cooks like a three-michelin-star-chef and gathers his team around the table to have lunch and relax. I do not want to miss the chance to express my gratitude to all the Danfoss engineers that helped me during my research stage, and for those who participated into the evaluation sessions even though their agendas were full: Marcus, Hauke, Helene, Martin, Kent, Henning, Christian, Klaus, Karl, Subhamoy and Supriya.

I want also to thank Danilo Beuche, from Pure-systems company, for promoting one of our research prototypes (*CustomDIFF*) in the newsletter of the pure::variants software release.

My deepest thanks to both Don Batory and Roberto Erick Lopez-Herrejón, for acting as external reviewers, and for their interest on discussing the the ideas presented in this Thesis.

Rightly, my family deserves a special mention. I will forever be grateful to my parents, Miren Karmele Mendizabal and Jesús Manuel Montalvillo, and my granny "amama Juani". They have raised us (my sister and me) in values such as, respect, humbleness, kindness, perseverance, and diligence. But more importantly they have raised us in love, and they have always encouraged us in every decision we made. To my sister Adriana, for always drawing a smile on my face when we connect via Skype. To my beloved Jokin, for his unconditional love, his support, and for his willingness to discuss my research with him. Thank goodness you are a PhD in software engineering and you can understand me :-) I want also to thank my friends, "nire kuadrila", for being there for me, for understanding my absences, and for just being yourselves and making me laugh every time we meet.

Finally, I want to thank the Basque Government and the University of the Basque Country (UPV/EHU) for the economical support I have received during the years 2014 to 2018, without which this Thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This chapter provides the reader with an overview of the Thesis. Section 1.2 contextualizes the Thesis, while Sections 1.3, 1.4, and 1.6 describe the problems that this Thesis tries to solve. Finally, Section 1.7 introduces the research methodology followed in this Thesis.

1.2 Context

Software Product Lines (SPL). SPL engineering has gained considerable momentum. Companies such as Boeing, Bosch, General Motors, Philips or Siemens resort to SPLs to broaden their software portfolio, increase return on investment, shorten time to market, and improve software quality[1] [CN01a, vdLSR07, Wei08]. In essence, SPLs aim at supporting the development of a whole family of software products through a

[1] Refer to http://splc.net/hall-of-fame/ for a list of successful SPL examples.

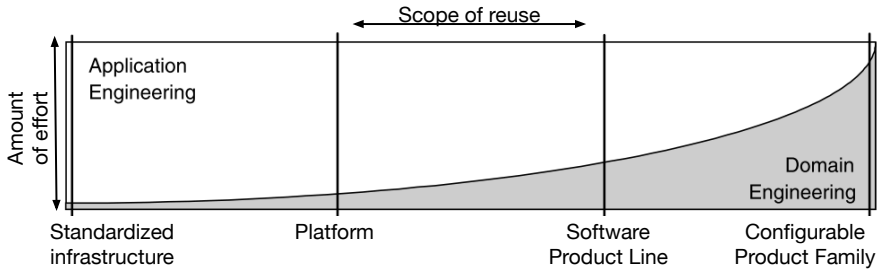

Figure 1.1: SPL maturity stages: from less mature (left) to more mature (right)[DSB05]).

systematic reuse of shared assets [CN01a]. To this end, SPL development is separated into two interrelated processes: (1) domain engineering (DE), where the scope and variability of the system is defined and reusable core-assets are developed; and (2) application engineering (AE), where products are derived by selecting and resolving variability [PBvdL05a, vdLSR07, CN01b].

In order to obtain the full benefit, SPL engineering aims at automating product derivation out of the reusable core-assets. These assets are characterized in terms of features, i.e. distinctive user-visible aspects, qualities, or characteristics of the SPL products. Ideally, product derivation is limited to indicating the set of features to be exhibited by products (so called "product configuration"), with no additional need for product development ("product customization" hereafter). In this scenario, products are obtained through a fully "Configurable Product Family" (refer to Figure 1.1), where almost the 100% of the effort is dedicated by domain engineers to the building of reusable assets, and, product customization does not exist. However, real scenarios might be far from ideal.

SPL Evolution. First, reaching a "Configurable Product Family" is hardly a one-shot effort but rather the result of a many year-long journey [KJK+06]. As shown in Figure 1.1, this might require intermediary stages (e.g. "Platform", "Software Product Line"), in which the reusable core-asset base does not fully support products' needs, and hence, application engineers need to develop the remaining functionalities themselves. Indeed, experiences from industry revealed that SPL adoption is frequently initiated with a partial core-asset baseline release that is first refactored from (a subset of) existing product variants, and gradually enlarged with newer functionalities extracted from products derived from it (e.g. [JB09, KST+14, TFC+09]). These newer functionalities are made available in the next SPL release, i.e. the set of core-asset, tested and ready to be reused by application engineering teams, from which newer products can derived and existing ones updated with newer functionality. Herein, a critical decision is the pace at which these SPL releases are made available. Commonly SPL releases come at heartbeats, i.e. regular intervals (e.g. twice a year) [Bre, GSLC14]. The benefits of heart-beaten releases is that released core-assets are supposed to work together. The downside is latency: it is not until the next SPL release that products can benefit from the core asset new versions. Similarly, it is not until the next SPL release that any of the product customizations promoted to core assets can be reused by other products. This might hinder products' time-to-market.

Second, even if SPLs have reached a "Configurable Product Family" level, SPLs might reach an scale and complexity that make them infeasible to evolve in a short time frame. Hence, when an organization aims to react to market events or urgent customer requests, strategies are needed to support fast adaptation, e.g. with product-specific extensions [DSB05, Jen07, Sch06a]. The bottom line, is that product customization can not always be avoided. Either because the SPL is on an "intermediary stage", or because the "SPL environment" forces the SPL to react faster through product customization. In this sense, highly customized products might be the symptom of not-yet fully mature SPLs, or volatile markets. Yet customized products uncover the potential of future SPL features, e.g. pointing to new customer requirements or new market niches. The question would be how to drive the evolution of the SPL taking into account that both core-assets and products need to co-evolve. The answer very much

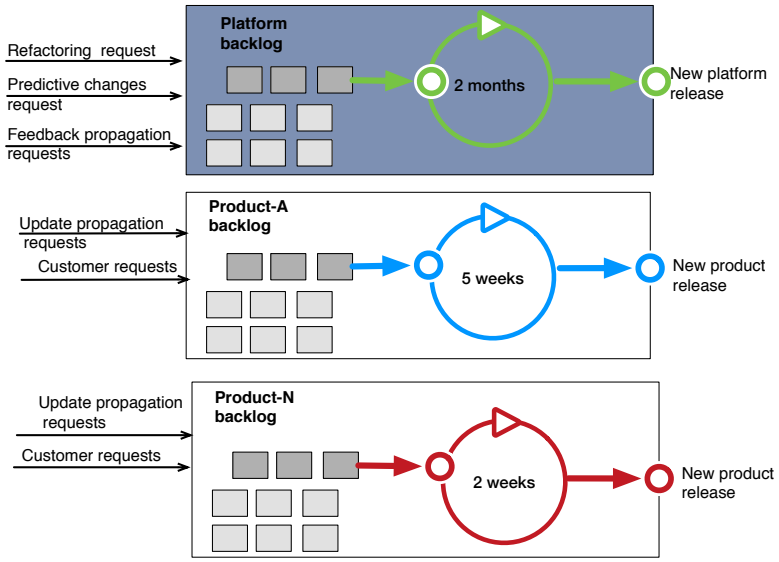

Figure 1.2: Grow-and-prune process' main operations: *growing* might be due to customer requests, while *pruning* might involve both feedback and update propagations.

depends on the SPL evolution model.

The grow-and-prune model. This model distinguishes between two stages during SPL evolution: *grow* & *prune*. The model allows products to *grow* once derived from the SPL core-asset base, and later *prune* the (product) functionalities deemed useful by refactoring and merging [FV03]. In this setting, both core-assets and products co-evolve by means of two paths: while product code is progressively ported to the core-asset realm following the *feedback path*, products can also be upgraded with newer functionalities and bug-fixes released by domain engineers following the *update path*. Figure 1.2 illustrates this scenario where it is easy to guess the rise of eventual tensions between domain engineers in the pursuit of quality and re-use effectiveness, and application engineers who are pushed by time-to-market and customer pressures. In this context, some issues arise. The next Section provides a general overview of the problem.

1.3 General problem overview

Evolving SPLs with the *grow-and-prune* model is challenging. The key is to find a balance between the right amount of growth and pruning [FV03]. Support for the *grow-and-prune* model should be given along the steps in the change/evolution mini-cycle proposed by Yau et al. [YCM93]:

1. **Identify change.** This step deals with identifying product customization. This requires to keep track of the customization effort involved in adapting the reusable core-assets for product-specifics, so that engineers can afterwards perform the required analyses in order to know how exactly derived products are changing the reusable core-assets; which are the most changed core-assets; and which are the products most customized. Hereafter, we refer to this practice as "customization analysis". The main problem is that customization analysis is time-consuming and error-prone, due to the lack of tools to provide a holistic view of product customization in terms of SPL concepts, i.e. "product" and "feature".

   (a) This Thesis aims at aiding engineers to perform customization analysis. Section 1.4 delves deeper into the problem statement of this issue.

2. **Analyze and plan change.** This step deals with analyzing which of the previously identified product customizations (the growth) deserves to be promoted to the core-assets base, and when it should be made. This requires to balance between the costs of refactoring and merging product customizations into the core-asset base, and the benefits of having them as reusable assets. Herein, cost models, impact analysis and risk assessments should support the decision.

   (a) This Thesis does not aim at contributing to this issue.

3. **Implement change.** This step deals with enacting the pruning, i.e. propagating changes between core-assets and products to keep them synchronized. Product customizations are pruned into the core asset base through feedback propagations, while products can also be upgraded with newer functionalities and bug-fixes through update propagations. Here, two issues arise:

   (a) merging and refactoring of product customizations into the core-asset base is difficult and time-consuming (a.k.a merge problem), due to the large number of conflicts that arise when disparate product developments are merged together.

      i. This Thesis aims at lessening the chances for merge conflicts. Section 1.5 delves deeper into the problem statement of this issue.

   (b) propagating changes between core-assets and products is time-consuming and error-prone, due to the lack of adequate tools

      i. This Thesis aims at aiding engineers on keeping both parties in sync. Section 1.6 delves deeper into the problem statement of this issue.

4. **Verify change.** This step deals with verifying and validating that propagated changes due to the pruning, do not affect the SPL products in an unexpected way. This would require to run regression tests on all the affected products.

   (a) This Thesis does not tackle this issue.

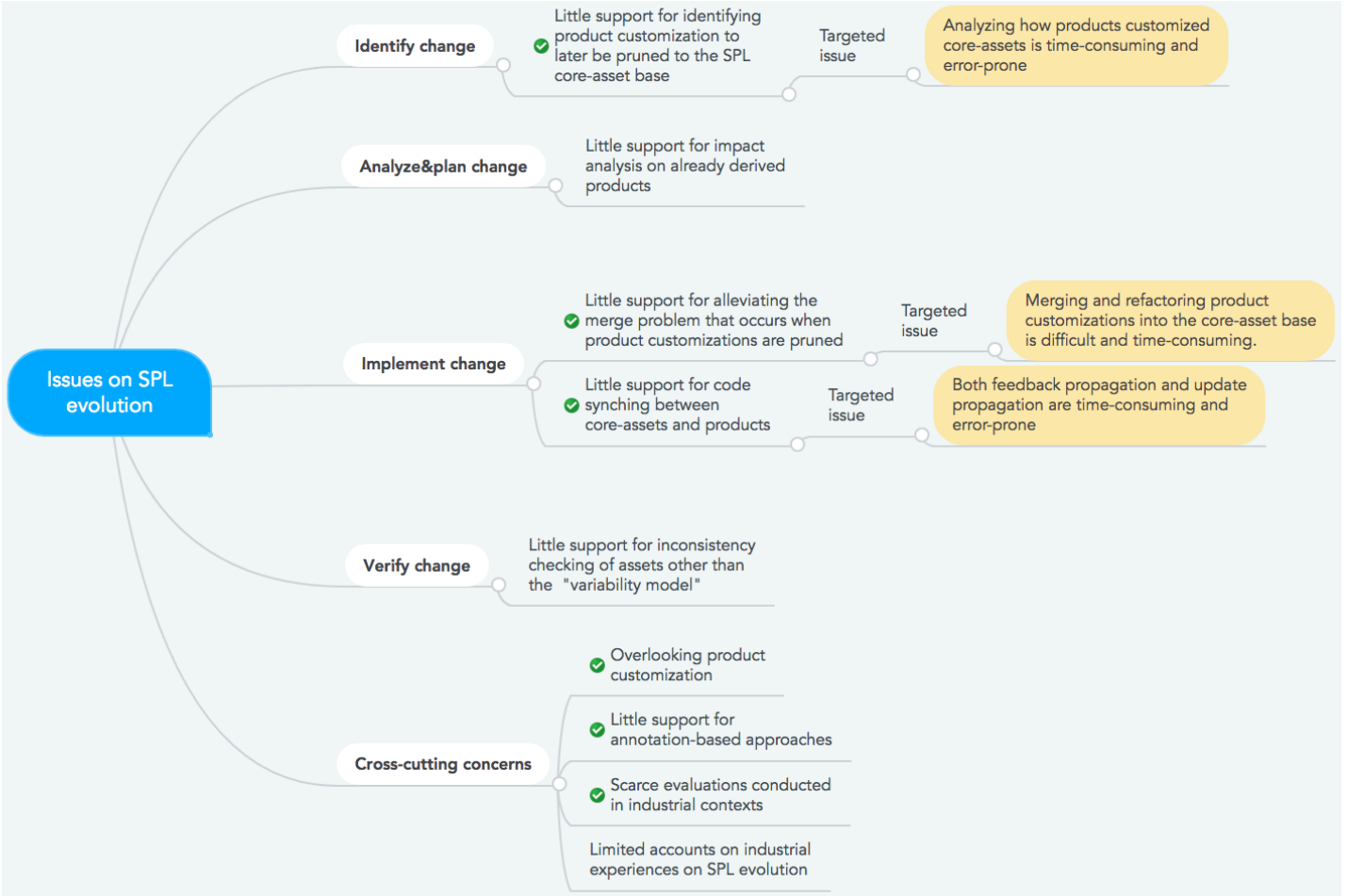

Figure 1.3: What the SPL literature on SPL evolution is not solving for the *grow-and-prune* model. Nodes with a green check mark (✔) are tackled in this Thesis. Refer to chapter 2 for a detailed literature review on SPL evolution. The map is online at *MindMeister* https://tinyurl.com/y7sdyb7w.

From the evidences gathered from a Systematic Mapping Study (SMS) we conducted, we could conclude that the state of-the-art in SPL evolution provides little support for SPL evolution in a *grow-and-prune* setting (refer to chapter 2 for a detailed account on the SMS). Figure 1.3 depicts the concerns not solved by the SPL evolution research in a *grow-and-prune* setting. Concerns are arranged along the steps in the evolution mini-cycle (nodes with white background). Those nodes with a green check mark (✔) are those investigated in this Thesis. The yellow nodes state the problem statements tackled by this Thesis. Specifically, we abound on the "identify" and "implement" change steps.

The next two Sections delve deeper on the two problems addressed by this Thesis. Methodologically, we resort to Design Science Research (DSR). DSR addresses design science problems (see Section 1.7), which tackle the design of artifacts to interact with a real world problem context in order to improve something in that context. First, we resort to root-cause analysis to methodically identify and correct the root causes of a problem. Second, when formulating design problems, we resort to Wieringa's [Wie14] template:

> ***Improve*** *<a problem context>*
> ***by*** *<(re)designing an artifact>*
> ***that*** *<satisfies some requirements>*
> ***in order to*** *<help stakeholders achieve some goals>*

The template provides information about what context is going to be improved, by the design of which artifact, such that a set of requirements are fulfilled, in order to meet stakeholders' goals. To exercise the template with an example, take the design problem of planning routes for aircraft taxiing on airports [tM10]:

> ***Improve*** *taxi route planning of aircraft on airports*
> ***by*** *designing multi-agent route planning algorithms*
> ***that*** *reduces taxiing delays*
> ***in order to*** *increase passenger comfort and further reduce airplane turnaround time*

For each of the problems tackled in this Thesis, we provide: (1) the context and key definitions, (2) the root-cause analysis of the problem to be solved, (3) the design problem formulated along Wieringa's template, (4) the set of research questions to be addressed, and finally, (5) the contributions.

1.4 Problem statement for "identify": analyzing product customization

1.4.1 Context & definitions

Following the *grow-and-prune* model, products can *grow* to meet customer changing needs or to resolve urgent bug-fixes. This growth can be achieved by adapting the core-assets from which products were derived, or by creating brand-new product specific

assets. In the context of this Thesis, we refer to this growth as "product customization", or just, "customization". Hence,

product customization takes place during product derivation, and refers to the process of changing the core-assets from which products were derived from, or create brand-new assets, in order to meet customer needs, or to resolve urgent bug-fixes.

Eventually, product customization needs to be pruned, by refactoring and merging into the core-asset base. The pruning is initiated with the "**identify change**" step, to identify product customization. This requires SPL engineers to elucidate whether (and which) products have customized the core-assets they were derived from, and analyzing how. Herein, a new range of questions might arise: how much effort are product developers spending on product customization?; how can customizations be promoted to core-assets?; which are the most customized core-assets?; to which extent is core-asset code being reused for a given product?; etc. This requires to look at the differences between core-assets and namesake assets once customized by products. In the context of this Thesis, we refer to this practice as "customization analysis". Hence,

customization analysis is the practice by which SPL engineers analyze how products have changed the core-assets they were derived from. Customization analysis is intended to help SPL engineers identify interesting customizations to be promoted to reusable core-assets for the next core-asset release.

The following Section introduces the problem that rises within this context, and its root-cause analysis.

1.4.2 Root-cause analysis

Figure 1.4 depicts the below-mentioned root-cause analysis as a mind map. The reader is encouraged to interact with the mind map online at https://tinyurl.com/yay46us8. The nodes can be unfolded to uncover the supporting evidences for each of the claims.

Problem statement

- Analyzing how products customized core-assets is time-consuming and error-prone.

Cause

- **Large number of files to be reviewed.** Anastasopoulos et al. [Ana09] provide a list of the steps that engineers should manually perform in order to know if any product has changed a given core-asset. Herein, traditional DIFF utilities are helpful, as they help engineers spot the differences between the core file and the same file once customized by a product (e.g. [FV03, SSRS16]). However, this one-diff-at-a-time approach can hardly scale up to SPLs, where both products and core-assets can easily account for hundreds of files.

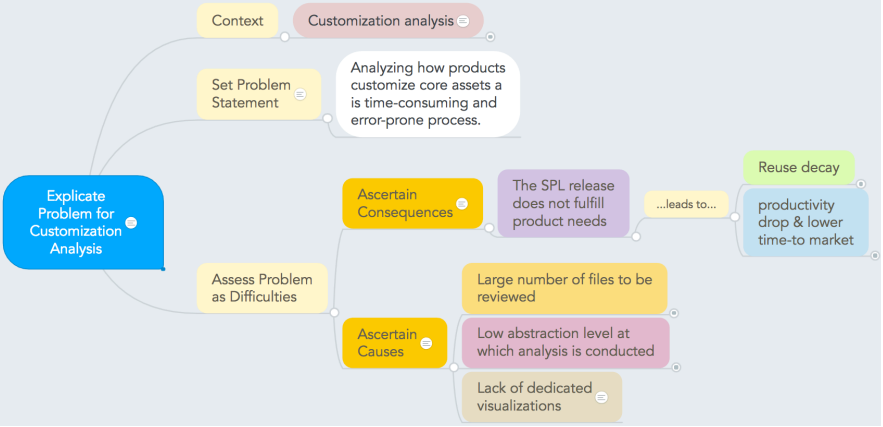

Figure 1.4: Mind map depicting the root-cause analysis for customization analysis. Interact with it online at https://tinyurl.com/yay46us8.

- **Low abstraction level at which analysis is conducted.** Not only engineers face a "bunch" of *diffs* to analyze, but this analysis requires to be done at higher levels of abstraction. Faust et al. [FV03] briefly reported how the SPL engineers at Deutsche Bank developed an script that computed how much code was specific to a product (measured by the lines of code). However, as the authors themselves recognize, this was not sufficient for engineers, and they ended up manually inspecting the code. Files are an implementation notion. By contrast, "*product*" and "*feature*" are abstract notions that are fleshed out through files. Analysis should then abstract away from *files* and be rephrased in terms of "*product*" and "*feature*".
- **Lack of dedicated visualizations**. Traditional diff utilities do not scale up to the potential thousand of changes that might be involved in a SPL. Appropriate tools are needed, not only to compute product customizations at feature and product abstraction levels, but also capable of "properly" visualizing these insights.

Consequences The pressure to deliver a new SPL core-asset release, together with an overwhelming set of customizations to analyse, risk the next SPL release not to fulfill the product needs (i.e. the right set of customizations will not eventually be pruned). If the next SPL release does not fulfill product needs, following consequences occur:

- **Reuse decay.** Products would perform more product customization, and reuse less from the SPL core-asset base. Products would then deviate from the SPL core-asset base, which risk a reuse decay [CKM+08][NNK16], i.e. a situation where products have degenerated that much from the SPL that products no longer reuse assets from the SPL, as they are not deemed useful anymore; or if deemed useful, integrating them into the product is so costly that it does not compensate

to reuse it (due to the high number of conflicts between the new core-assets and the customized code).

- **Productivity drop and higher time-to-market.** If application engineers start to rely more and more in product customization, this would directly lower down their productivity, and would in turn increase the time-to-market of the products.

How can we help with the problem of *"analyzing how products customized core-assets is time-consuming and error-prone*"? Next Section provides the design problem along Wieringa's template.

1.4.3 Design Problem

Design problems assume a *context* and stakeholders *goals*, and call for an *artifact* such that the interactions of (artifact × context) help stakeholders to achieve their goals [Wie14]. Our design problem formulated along the lines of Wieringa's template could be described as follows:

> **Improve** *customization analysis*
>
> **by** *designing a data warehouse approach*
>
> **that** *satisfies scalability & usefulness (as for satisfying engineers' information needs)* **so that** *SPL engineers can effectively conduct the "identify" step during SPL evolution*

This template reads as follows. The *context* to be improved would be "customization analysis", and the *goal* to achieve would be for SPL engineers to effectively conduct the "identify" step during SPL evolution, i.e. identify all the customizations that happened in products. To this end, we advocate for the design of an artifact: a data warehouse (DW). This purposeful artefact needs to address a set of requirements. First, the artifact must be scalable, as product customizations can account for hundreds of files across hundreds of different products. In this sense, DW approaches are well known to be capable of dealing with big volumes of data. Second, the design of the data warehouse needs to fulfill the "information needs" required by the SPL engineers when conducting customization analysis (e.g. "which are the most changed features by the products?","how has the product PA customized the feature FA"). Gathering the data for these information needs might require to access heterogeneous data from different and multiple information systems. To this end, raw data is conducted through an Extract, Transform, Load (ETL) process that ends up being arranged in a star schema, which accounts for facts (i.e. the aspects to be measured) and dimensions (i.e. the ways measures are going to be broken down). Finally, product customization needs to be visually depicted. When product customization accounts for hundreds or thousands of records, information can be better understood with visual representations.

On these grounds, we believe DW techniques might tackle *"satisfies scalability & usefulness"* for improving *"customization analysis"*. Next Section lists the set of research questions (RQs) addressed in this Thesis.

1.4.4 Research questions

The use of DWs for customization analysis raises a set of research questions (RQs). These are listed next.

RQ1: Which are the information needs for customization analysis? For each information need, how much time is needed to get it?

By investigating this RQ, we aim at obtaining the set of information needs required by the SPL engineers when conducting customization analysis (e.g. "which are the most changed features by the products?", "How has the product PA customized the feature FA"). These are the requirements that need to be met by our DW approach. We also look into how much time SPL engineers need to get those "information needs" when performing customization analysis.

RQ2: To what extent can previous information needs be satisfied through a data warehouse? If so, what would its star schema look like?

By investigating this RQ, we aim at designing a DW approach that captures the information needs previously identified. The star schema needs to be designed to support the analysis of customization (i.e. facts) at different levels of abstraction (i.e. dimensions).

RQ3: How can customization analysis be visualized?

By investigating this RQ, we aim at visually representing product customization to easy customization analysis at a glance.

1.4.5 Contributions

This Thesis aims at contributing to the previous research questions as follows:

RQ1. We elaborate on the information needs that arise during SPL evolution scenarios ("feature evolution" and "product evolution"). The importance of each information need, and the required time to get them are validated through a questionnaire delivered to SPL practitioners.

RQ2. We developed *CustomDIFF*, a DW approach to customization analysis that uses *Git* as the operational system from where fact data is obtained, and *pure::variants* as the SPL framework. The designed star schema allows SPL engineers to aggregate customization facts along different levels of abstraction, such as, product, feature, core-asset and component. CustomDIFF has been tested with SPL practitioners to evaluate its usefulness and use of use for customization analysis.

RQ3. We resort to Alluvial diagrams to visualize the customization effort at a glance. These diagrams are a type of flow diagrams. Here, the flow stands for the customization effort that goes from core-assets to SPL products where customization was needed.

1.5 Problem statement for "implement": peering into peers

1.5.1 Context & definitions

The pruning requires that those product customizations deemed useful are integrated into the core-asset base by merging and refactoring [FV03]. Note, that due to the previous "grow" phase, products might have diverged to much from each other. During the pruning domain engineers need to reconcile these divergences by resolving the conflicts that arise from merging together disparate product customizations. The higher the divergences between the products, the higher the conflicts and the complexity to resolve them. When the time to resolve these conflicts exceed the time needed to perform the original changes, we are in the so-called *merge problem* situation [Duv07] (a.k.a integration hell or merge hell). Hence, the

merge problem arises during the pruning stage, when disparate product customizations are merged into the core-asset base resulting in a multitude of conflicts, whose time to be resolved exceed the time it took to make the original changes.

Our hypothesis is that providing application engineers integrated support for *looking into other products' code* right during product development, promotes early reuse across products and small refactoring improvements, in the search lessening the conflicts of merge problem that occurs during the pruning. We refer to this practice as "code peering". Hence,

Code peering (or peering) refers to the practice that takes place during product development, whereby product engineers inspect and compare other products' code with their own code, and if interested, merge them together. Code peering is intended to promote early reuse of product developments across product teams, with the aim of lessening the merge problem during pruning.

The next Section delves deeper into the problem and its root-cause analysis.

1.5.2 Root-cause analysis

Figure 1.5 depicts the root-cause analysis as a mind map. The reader is encouraged to interact with the mind map at https://tinyurl.com/y9fqucpe. The nodes can be unfolded to uncover the supporting evidences for each of the claims.

Problem statement

- Merging and refactoring product customizations into the core-asset base is difficult and time-consuming.

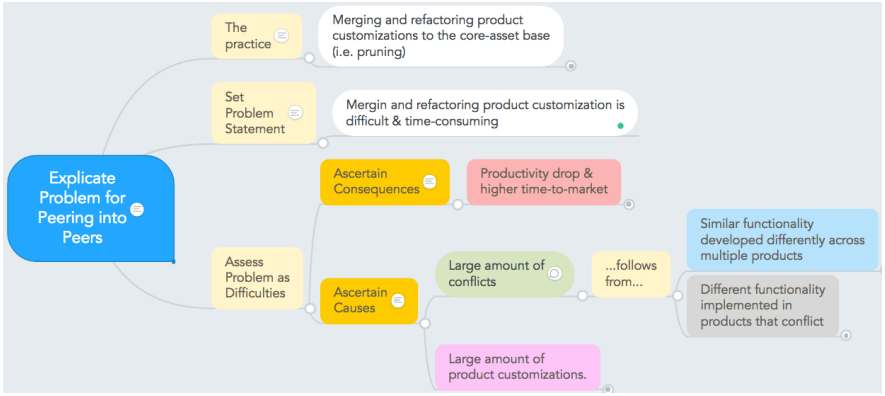

Figure 1.5: Mind map depicting the root-cause analysis for peering into peers. Interact with it online at https://tinyurl.com/y9fqucpe.

Causes

- **Large amount of conflicts.** During the *growth* phase, products detach from the core-asset baseline, and follow their own life-cycle, without paying attention to what other product teams are developing. The more the products deviate from each other during the growth phase, the higher the chances for conflicts during the merge. These conflicts arise in cases where across product teams, the same functionality is implemented multiple times (but differently in each product) [DSB05, TMMK11]. Afterwards, domain engineers need to reconcile these different implementations into a single one that subsumes all of the others. Additionally, in cases where different functionalities are implemented across different products, conflicts also arise when these all need to be merged. What is needed is a way so that product teams do not deviate that much from each other, hence, facilitate the later merging. The difficulty of the merge conflicts, and hence, the time needed to resolve them is influenced by (1) the complexity of the conflicting lines, (2) the knowledge of the developers on the conflicting code, (3) the complexity of the files with conflict, and (4) the number of conflicting lines [MNSD17].
- **Large amount of product customizations.** The higher the number of products and product customizations, the higher the the chances for conflicts when these are merged together.

Consequences

- **Productivity drop & higher time-to-market**. The integration work and effort for porting product customizations into the core asset can become a major part in the DE teams work load, if there are many product customizations to prune, and if these cause a large amount of conflicts to resolve. This reduces the time

for DE to work on feature improvements and new feature requests [JB09]. This paces down the next core-asset baseline release, compromising product's time-to-market.

How can we help with the problem of *"merging and refactoring product customizations into the core-asset base is difficult and time-consuming"?* Next Section provides the design problem along Wieringa's template.

1.5.3 Design Problem

Our design problem formulated along the lines of Wieringa's template [Wie14] is as follows:

Improve *the merge problem*

by *leveraging Web Augmentation, Data Warehouse and 3-way comparison techniques for code peering*

that satisfy *respect focus* and *compatibility*

so that *the chances for conflicts are lessen and SPL engineers can effectively conduct the "implement" step during SPL evolution*

This template reads as follows. The *context* to be improved would be "the merge problem" that arises during the pruning of product customizations, and the *goal* to achieve would be for SPL engineers to effectively conduct the "implement" step during SPL evolution, i.e. merging and refactoring product customizations. To this end, we advocate for leveraging Web augmentation (WA) [DA15], Data Warehouse (DW), and 3-way comparison&merging techniques to provide peering functionality. Web augmentation permits third parties to adapt Web sites, data warehouse techniques enable making better and faster decisions [KR02], and 3-way comparison&merging techniques help engineers compare&merge two versions of a file while also considering the origin of both files (a.k.a. *common ancestor*) [3waa].

In this Thesis, we utilize web augmentation techniques to enhance Github, the most popular web-based Git-based Version Control System (VCS) repository hosting service, with *a peering bar* that makes product teams aware of what features are other product teams currently customizing. This bar brings product engineers into a DW solution, when clicking on it. This web-based DW solution permits product teams to have an overview on how much are other product customizing the features their product is reusing. Finally, the DW solution acts as a 3-way comparison&merging enactor, that permits product teams to compare their product's code with other products' code, for a given feature, and merge them if wanted. This cross-product peering and reuse lessens the deviations between products, and as consequence, would lessens the conflict occurrence during the *pruning* phase for domain engineers. Hence, this Thesis proposes the use of both WA, DW, and 3-way comparison techniques during the *grow* phase, so that SPL engineers can effectively conduct "*implement change*" step during the prune phase.

Although, *code peering* encourages easy pruning, this might be an ancillary activity from an AE perspective, as for them, product development comes first. This has a main

implication: *code peering* should "respect the focus" of application engineers, i.e. do not interrupt product development. Additionally, support for *code peering* needs to satisfy compatibility requirements, i.e. the extent to which the enhancement is aligned to previous user experiences [and82]. In our case, this experience refers to the usage of Github. Hence, the solution should be compatible with Github way of working. On this grounds, we believe that WAs techniques are good means for enhancing web-based Version Control System (VCS) tools for *code peering*, without compromising application engineers' focus on product development.

Next Section lists the set of research questions (RQs) addressed in this Thesis.

1.5.4 Research questions

This Thesis elaborates around three main research questions:

RQ1: How is the grow phase conducted in practice?

By investigating this RQ we aim at making explicit who, when and how do stakeholders participate during the *grow* phase. Although the *grow-and-prune* model is being referred to in the literature, the nitty-gritty details have seldom been reported.

RQ2: What are the characteristics of the merge problem in SPLs? And, how can we lessen the merge problem in SPLs?

By investigating this RQ we aim at identifying the characteristics that turn the pruning phase into a merge problem, and a possible way to lessen this.

1.5.5 Contributions

This Thesis aims at contributing to the previous research questions as follows:

RQ1. Description of the roles and interactions that intermingle in a *grow-and-prune* approach to SPLs, motivated by the Danfoss case.

RQ2. Characterization of the merge problem and how it differs from the merge problem that also appears in traditional single-system development. We propose a new practice, *code peering*, as a possible way to alleviate it. This begs the question whether it is worth diverting product developers' attention for the sake of making easier the subsequent pruning by domain engineers. Using the theory of Attention Investment [Bla02] as a narrative, we introduce four design principles that drive how code peering can be introduced for SPL development.

RQ2. A realization of these principles using GitHub as the VCSs, and *pure::variants* as the SPL framework. As a *proof-of-concept* we developed *PeeringHub*, a tool that supports code peering through: (1) a Chrome extension that enhances GitHub with *peering bars* that provide brief information about what features are other peers changing, (2) a web-based application that provides alluvial-based high-level visualizations indicating the features available for *code peering*, and (3) *feature-based* 3-way comparisons so that product engineers can analyze how a given product is changing the code of a given feature w.r.t its own.

1.6 Problem statement for "implement": synchronizing core-assets and products

1.6.1 Context & definitions

Enacting the pruning stage requires propagating changes between core-assets and products, so that both parties are synchronized. This introduces two **sync operations**: the update propagation (from Domain Eng. to Application Eng.), and the feedback propagation (from Application Eng. to Domain Eng.) [Kru03]:

Feedback propagation is the process that serves for: extending the scope of the product line to emerging application engineering requirements [Kru03], as well as, incorporating bug-fixes resolved in products [FSK+16]. The integration of these changes into the core-asset base may require updates to be applied to already existing products [Kru03].

Update propagation is the process that serves for: configuration repair (synchronize products configuration when variability model changes) [BM14], as well as, product upgrade (where latest versions of reusable assets are propagated to products) [Kru03]. In the latter case, for every product derived from the original core-asset, an update operation is required. If products have customized the core-asset then, the update operation may require a manual merge for each product [Kru03]. When to conduct the upgrade differs significantly for the different products in the SPL. While some tend to upgrade rather quickly, some do not upgrade for a long time, even when not close to the product's release [JB09].

In order to preserve a correct, complete and consistent synchronization between core-assets and products, Software Configuration Management (SCM) for SPL development needs to account also for these propagations. SCM is the discipline that enables engineers to keep control and track software changes (i.e. evolution). Product line SCM must support [CN01a]: (1) the derivation process of a product from the core-asset base, (2) update propagation process, (3) feedback propagation process. In a nutshell, SCM fulfill these requirements by relying on both (1) **tools** to track changes to software assets, i.e. Version Control Systems (VCSs), as well as, on (2) **policies** for engineers that establish when and how to branch, merge, and commit code (captured as branching models). However, the state of-the-art tools and practices are not tailored to SPL specifics, and this causes update and feedback propagations to be time-consuming and error-prone.

The next Section delves deeper into the problem and its root-cause analysis.

1.6.2 Root-cause analysis

Figure 1.6 depicts the root-cause analysis as a mind map. The reader is encouraged to interact with the mind map at https://tinyurl.com/ya777m2x. The nodes can be unfolded to uncover the supporting evidences for each of the claims.

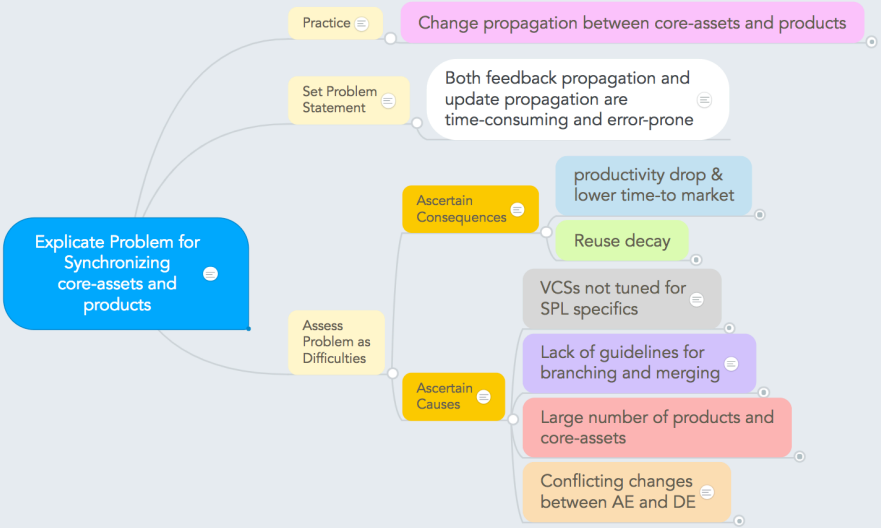

Figure 1.6: Mind map depicting the root-cause analysis for propagating changes between core-assets and products. Interact with it online at https://tinyurl.com/ya777m2x.

Problem statement

- Both feedback propagation and update propagation are time-consuming and error-prone.

Causes

- **VCSs not tuned for SPL specific operations.** VCSs are mainly thought for single-systems, and hence, they do not support product derivation, nor the update and feedback propagation paths required for SPL development [Ana13, TMN08, vGB02]. State-of-the-art VCSs, such as *Git/GitHub,* provide the basics but fall short in supporting sync operations between separated VCS repositories (e.g. a *CoreAsset* repository and a *Product* repository). All *GitHub* offers is the *fork* link to create a clone of a repository. However, forking (i.e. cloning) is not how products are derived. Indeed, products are built from a subset of core-assets while forking would entail copying the entire *CoreAsset* repository. Likewise, GitHub's *pull request* is also thought for synchronizing a whole repository, hence lacking a more piecemeal synchronization, i.e. at feature level. The same reasoning applies if product derivation is equated to *branching* instead as to *forking*. This lack of adequacy forces engineers to rely on workarounds, which are time-consuming and error-prone.
- **Lack of guidelines for branching and merging**. While VCSs are tools that track software development, branching and merging policies provide rules to

support the efficient synchronization of software development efforts. These rules are captured in the form of branching models. Extensive literature exists on branching models for single-system development, e.g. [ABCO98, WS02b, PSW11, Gitb]. In the case of SPLs, little details are given about how this is exactly done. This is unfortunate since the adequacy of branching models very much depends on the processes to be supported. Indeed, industrial experiences have reported how the lack of stablished branching and merging policies prevented engineers from synchronizing their developments, causing inefficiencies [NNK16][JB09].

- **Large number of products and core-assets.** When a core-asset is upgraded, existing products might need to get this upgrade. If products have customized the core-asset then, the update operation may require a manual merge [Kru03]. Since products reuse only a sub-set of the core-asset base, engineers would need to elucidate only the newer versions of the core-assets that the product is reusing need to be propagated. Hence, the process of syncing core-assets and products requires a big effort for SPLs that account for hundreds of core-assets and products.

- **Conflicting changes between DE & AE**. Syncing changes between core-assets and products can be a very time-consuming and error-prone process if the newer versions of core-assets conflicts with product customization. The longer both parties wait to sync, the greater the chance for conflicting changes. The appropriate time to conduct the upgrade differs significantly for the different products in the SPL. While some tend to upgrade rather quickly, some do not upgrade for a long time, even when not close to the product's release [JB09]. This risks synchronizations to be further postponed, which higher the change for conflicting changes, and for a time-consuming and error-prone integration process.

Consequences

- **Productivity drop and higher time-to-market.** If core-assets and products are not synchronized, products would not get the latest available bug-fixes and new functionalities [JB09]. Hence, products would need to develop these themselves, which makes application engineers less productive, and causes a higher time-to-market of products.

- **Reuse decay.** The less synchronized the products are with the core-assets, the more the products deviate from the core-asset base. This means that products would less frequently want to reuse assets from the core-asset base, as the updates would require more and more integration effort. The longer the update delay, the higher the amount of newly available updates, and the higher the conflicts between DE and AE developments. This risks the SPL to a reuse decay. Unless products are in sync with latest available updates, products will start clone-and-own outside the SPL and in the second case do not upgrade [JB09].

How can we help on the problem of "*both feedback propagation and update propagation are time-consuming and error-prone*"?Next Section provides the design problem along Wieringa's template.

1.6.3 Design Problem

Our design problem formulated along the lines of Wieringa's template [Wie14] is as follows:

> **Improve** *update propagation and feedback propagation*
>
> **by** *using Web Augmentation techniques to enhance Github to SPL practices*
>
> **that satisfy** *compatibility*
>
> **so that** *SPL engineers can effectively conduct the "implement" step during SPL evolution*

This template reads as follows. The *context* to be improved would be "update and feedback propagation process", and the *goal* to achieve would be for SPL engineers to effectively conduct the "implement" step during SPL evolution, i.e. synchronizing core-assets and products. To this end, we advocate for enhancing VCS tools by means of Web Augmentation (WA) techniques. Specifically, we leverage Github and extend its functionality with operations though for SPLs (i.e. product derivation, and update&feedback propagation). However, this enhancement, needs to satisfy compatibility requirements. Compatibility refers to the extent to which the enhancement is aligned to previous user experiences [and82]. In our case, this experience refers to the usage of Github. Hence, the solution should be compatible with Github way of working.

On this grounds, we believe WAs techniques are good means for enhancing web-based VCS tools for SPL specifics. Next Section lists the set of research questions (RQs) addressed in this Thesis.

1.6.4 Research questions

This Thesis elaborates around two main research questions:

RQ1: How can products and core-assets be arranged in VCSs (e.g. Git), and how does the branching model look like?

By investigating this RQ we aim at elucidating how core-assets and products, which are developed at different paces and by different teams, can be arranged under VCS repositories.

RQ2: How can VCSs' front-ends (e.g. Github) help on synchronizing core-assets and products?

By investigating this RQ we aim at providing synchronization operations as first class constructs using VCS basic operations (i.e. commit, branch and merge).

1.6.5 Contributions

This Thesis aims at contributing to the previous research questions as follows:

RQ1. We propose a VCS repository architecture, which distinguishes between the *CoreAsset* repository, where domain engineering takes place, and *Product* repositories, where product engineering occurs. We additionally provide branching models for each repository in which sync actions operate.

RQ2. We elaborate on the operational semantics for sync actions. The previous branching model permits sync operations to be expressed in terms of basic VCS constructs. This in turn implies that eventual mismatches that rise during synchronization are resolved *à la VCS*, i.e. highlighting *diff*-erence between distinct versions of the same artifact (traditionally, using the *diff* option in VCSs). Therefore, we do not aim at automatic sync. Our aim is much more humble: tap into VCS popular mechanisms for SPL engineers to achieve sync in a way similar to what they do for single products. However, this results in a conceptual gap between how sync paths are conceived, and how they are realized down into branching and merging. To close this gap, we propose leveraging VCSs with SPL sync operations.

RQ2. As a proof-of-concept, we developed *GitLine*, a browser extension for Firefox, that extends GitHub with SPL sync operations. Through a single click, product engineers can now (1) generate *Product* repositories from a *CoreAsset* repository, along a certain feature configuration, (2) update propagations of newer core-asset versions, or (3), feedback propagation of product customizations.

1.7 Research Methodology: Design Science Research

In this Thesis we followed Design Science Research (DSR). DSR is the scientific study and creation of artefacts as they are developed and used by people with the goal of solving practical problems of general interest [JP14]. Thus, design science is one approach to investigating artefacts.

DSR takes a problem solving instance, starting from problems experienced by people in practice, and then tries to solve them. It does so by creating, positioning, and repurposing artefacts that can function as solutions to the problems. The key differentiator between routine design and design research is the clear identification of a contribution to the archival knowledge base of foundations and methodologies. Design science is viewed mainly from an IT and information systems perspective. However, the principles underlying design science are applicable to many other areas [JP14].

Hevner [Hev07] proposes a three cycle process for DSR (see is Figure 1.7).

- **The Relevance Cycle** initiates the DSR by identifying and analyzing problems to be addressed in an context. The problem must be precisely formulated and justified by showing it's relevant within the context. The problem has to be of general interest and remarking causes to the problem might be identified and analyzed. Herein, the root-cause analysis provides a way to methodically

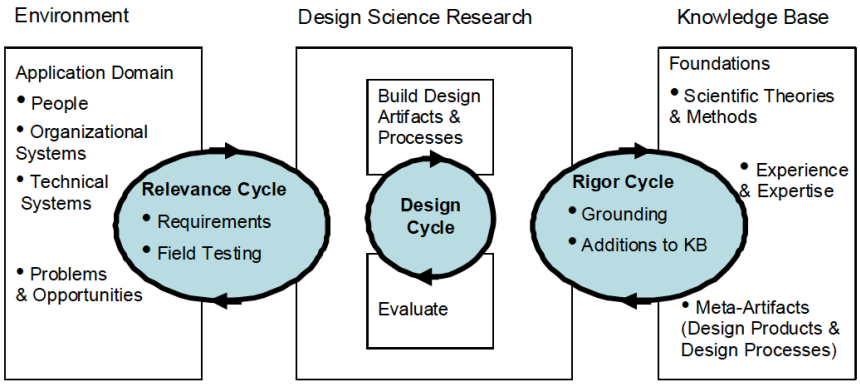

Figure 1.7: Design Science Research (DSR) Cycles (taken from [Hev07]).

identify the root causes of a problem. While the consequences of a problem illustrate the importance of the problem, the causes of the problem are the targets to be corrected/attacked for preventing the problem recurrence. The relevance cycle does not only provide the problems to be addressed, but also defines acceptance criteria for the ultimate evaluation of the research result, i.e. the requirements to be addressed by the solution in order to solve the problem.

- **The Design Cycle** is the heart of any design science research project. This cycle of research iterates between two main activities. First, the design and construction of an artifact that solves the problem by meeting the requirements identified in "The Relevance Cycle". And second, the evaluation of the artifact. The feedback obtained from the evaluations can make the artifact to be further refined.

- **The Rigor Cycle** connects the design science activities with the knowledge base of scientific foundations, experience, and expertise that informs the research project. The rigor cycle provides past knowledge to the research project to ensure its innovation. It is contingent on the researchers to thoroughly research and reference the knowledge base in order to guarantee that the designs produced are research contributions and not routine designs based upon the application of well-known processes [HMPR04].

This dissertation has been developed along the DSR hallmarks.

- As for the relevance cycle, we identified two problems in the context of evolving SPLs following the *grow-and-prune* model. Following a root-cause analysis, we analyzed the causes that lead to the problem, as well as, the consequences that they could generate. For each of the identified problems, we additionally identified the requirements that needed to be addressed in order to solve the issue.

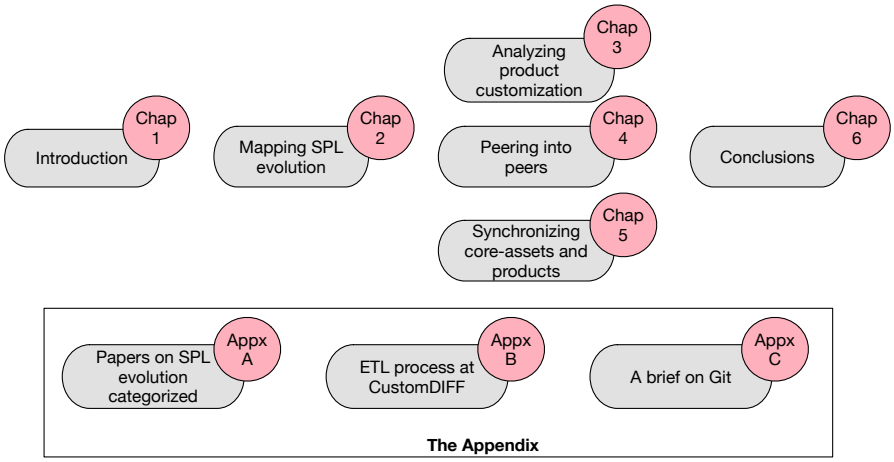

Figure 1.8: Chapter map.

- As for the design cycle, we designed and build two artifacts, *CustomDIFF* and *GitLine*, targeting each of the identified problem. Due to the difficulties in finding a company both willing to share its SPL set-up and letting us to evaluate these artefacts, only the former artefact has been evaluated into an industrial context.
- With respect to the rigor cycle, we have both nurture from the knowledge base, as well as, we have contribute to it. We achieved rigor by positioning our research in the context of evolving SPLs by the *grow-and-prune* model, and by applying existing foundations and best practices when designing and building both artifacts.

We believe to have contributed to the SPL knowledge through:

1. a mapping on SPL evolution literature,
2. a DW approach to customizations analysis: requirements & artifact (i.e. *CustomDIFF*)
3. a new practice, included into the application engineering process, that can help alleviating the merge problem in SPLs (i.e. code peering)
4. a repository branching model for SPLs and their operations counterpart for keeping both core-assets and products synchronized.

1.8 Outline

This section outlines the content of the Thesis. Figure 1.8 illustrates the chapters of this dissertation. Below, a summary of each chapter in this dissertation is provided.

Chapter 2 This Chapter presents a Systematic Mapping Study (SMS) on SPL evolution. It provides the reader with a background on SPLs, and maps the existing research on SPL evolution along four facets: evolution activity (e.g., identify, analyze and plan, implement), product-derivation approach (e.g., annotation-based, composition-based), research type (e.g., solution, experience, evaluation), and asset type (i.e., variability model, SPL architecture, code assets and products). The chapter ends with the identification of the two issues that this Thesis investigates.

Chapter 3 This chapter introduces the practice of customization analysis, i.e. the practice by which SPL engineers analyze how products have customized the core-assets after being derived from the SPL. In this chapter we propose the use of data-warehouse techniques for customization analysis. Requirement Analysis, Dimensional Modeling and Reporting Tools are discussed, that end up in *CustomDIFF*, a data warehouse tool that uses Git as the operational system and pure::variants as the SPL framework. This work has been motivated and validated in the context of Danfoss Drives, a SPLC-awarded hall-of-fame company.

Chapter 4. This chapter introduces the merge problem that arises during the pruning phase, i.e. when disparate product customizations are merged into the core-asset base resulting in a multitude of conflicts, whose time to be resolved exceed the time it took to make the original changes. In this chapter, we propose *code peering* practice, intended to promote early reuse across product teams during the grow phase, with the aim of lessening the subsequent merge problem. We introduce four design principles that drive how code peering can be introduced for SPL development. We present a realization of these principles, *PeeringHub*, that works for Git/GitHub, and SPLs developed with *pure::variants*.

Chapter 5. This chapter introduces the practice of synchronization between core-assets and products. In this chapter we propose the use of Version Control Systems (VCSs) to aid on such synchronization using using traditional VCS constructs (i.e. merge, branch, fork and pull). We discuss implications for branching models for SPL development, and provide sync operations as a first-class constructs. We present a browser extension, *GitLine*, that extends GitHub with sync operations for SPLs.

Chapter 6. This chapter concludes the Thesis by remarking main results, listing the publications that endorse this Thesis, enumerating the limitations of the current solutions, and suggesting possible future work.

Appendix A. This Appendix serves chapter 2. It provides the list full list of the papers that were included in the SMS, categorized on

Appendix B. This appendix serves chapter 3. It provides the algorithms for the *Extract Transform and Load* (ETL) process followed in *CustomDIFF*.

Appendix C. This appendix serves chapter 4 mainly, although it may be of interest for chapter 3 too. It provides the reader with a brief on git, its basic operations, and points to popular branching models for the development of single-systems.

1.9 Conclusion

The intention of this chapter was to give an overview of the contents of this dissertation. We introduced the context that frames this Thesis, as well as, we defined the problems that it tries to solve. The contributions to these problems were also listed. Finally, the research methodology followed in this Thesis was briefly introduced.

The next chapter provides a mapping study on SPL evolution.

Chapter 2

Mapping Software Product Line Evolution

2.1 Overview

This Chapter[1] presents a Systematic Mapping Study (SMS) that maps the existing research on the area of SPL evolution. Note, that in the context of this Thesis SPL evolution is achieved by co-evolving both core-assets and products. However, this might well not be the case for other research efforts, that address solutions for SPLs at other reuse levels (e.g. SPLs that do not consider products to evolve). Hence, this SMS maps studies on the area of SPL evolution, idenpendently of the SPL maturity level the study is addressing.

This Chapter provides the reader with a brief background on SPLs, and describes the characteristics that makes SPL evolution challenging. More importantly, the existing research on SPL evolution is mapped along main four facets: evolution activity (e.g., identify, analyze and plan, implement, verify), product-derivation approach (e.g., annotation-based, composition-based), research type (e.g., solution, experience, evaluation), and asset type (i.e., variability model, SPL architecture, code assets and products). Analyses of the results indicate that "Solution proposals" are the most common type of contribution (31 %). However, few studies do address solutions for co-evolving core-assets and products. The Chapter ends with the identification of the two issues that this Thesis investigates.

Next we introduce the motivations and research questions behind the SMS.

2.2 Introduction

As the SPL domain matures, evolution concerns come into play [Bos02, DNGR08]. Unfortunately, the term “evolution” has long been recognized as being overloaded with diverse matters [BR00]. For the purpose of this work, “evolution” refers to

[1] The content of this Chapter has been previously published in [MD16]

the adaptation of the SPL as *a result of changing SPL requirements*. From this perspective, evolution is triggered by requirement changes, and not so much by refactoring. Evolution happens as a result of SPLs moving from adoption to maturity. In their infancy, SPLs strive to fix defects. At adulthood, SPLs might have less defects, but their wider customer base more likely increases the chances for new functionality requests.Indeed, SPLs' long life-span makes evolution a top priority, yet far from being fully resolved [BP14]. SPL characteristics that make evolution specially challenging include: (1) separation of development into Domain Engineering (DE) and Application Engineering (AE), (2) existence of assets of different types of variability and abstraction, and (3), high number of interrelations between assets [Mcg03, AK08, DSB05]. One of the first works (conceptually) addressing evolution in SPLs is [SB99]. Svahnberg et al. analyze the life-span of two industrial SPLs, and classified SPL evolution according to common scenarios that arose during evolution ("requirement evolution", "architecture evolution", and "component evolution"). Thereafter, few efforts have been made to gather studies addressing this issue. Two exceptions are [BP14, Mcg03]. The most referenced work is McGregor's one who introduces basic evolution concepts and discusses practices that initiate, anticipate, control, and direct the evolution of SPL assets [Mcg03]. Botterweck et al. [BP14] present the most recent summary on the topic. Authors provide an overview on three main issues: migration to SPLs, planning SPL evolution, and implementation of SPL evolution. However, none of the previous works systematically review the existing literature, and thus, they do not provide coverage of the different topics.

A systematic mapping study is an evidence-based approach where existing works can be categorized, often giving a visual map of its results [KC07, PFMM08a]. This work presents the outcome of such approach conducted for the literature on SPL evolution available up to July, 2015 which resulted in 107 primary studies. The overall research questions follow:

> **RQ1:** What types of research have been reported, to what extent, and how is coverage evolving?
>
> **RQ2:** Which product-derivation approach received most coverage, and how is coverage evolving?
>
> **RQ3:** Which kind of SPL asset received more attention, and how is attention evolving?
>
> **RQ4:** Which activities of the evolution life-cycle received most coverage, and how is this coverage evolving?

Answering RQ1 would allow us to assess maturity within the field, e.g., if research is limited to solution proposals or rather it takes a step forward and conducts some kind of validation, or even better, it evaluates the solution in industry. On the other hand, RQ2 would allow us to assess how SPL product derivation approaches are catching on. Next, RQ3 looks at *"the subject"* of evolution, i.e., the SPL asset being subject to change. This includes the variability model, the SPL architecture, code assets or SPL's products. Conversely, RQ4 looks at *"the verb"* of evolution, i.e., which evolution tasks authors have focused on (e.g. identify change, analyze change, implement change, verify change). In summary, the outcome of this study might help to identify trends,

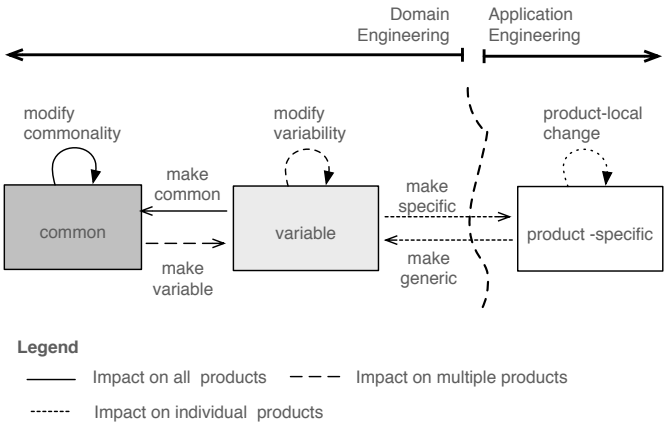

Figure 2.1: Types of changes (based on [BP14, Kla08]).

hotspots and gaps both in terms of "the verb" and "the subject" of SPL evolution. Also, a brief is provided for each of the 107 primary studies. Special effort is dedicated to arrange these studies within a fine-grained schema that might help newcomers to better pinpoint the area of interest.

The remainder of this Chapter is organized as follows. Section 2.3 provides an overview on SPLs, highlights what makes SPL evolution challenging, and points to previous mapping studies in the SPL field. Section 2.4 describes the systematic methodology used to conduct this mapping study. Section 2.5 provides an annotated bibliography that serves to map primary studies into a finer-grained classification of the evolution activities. Section 2.6 analyses the results of the mapping, and answers the RQs. Conclusions end the Chapter.

2.3 Background

This section provides an overview on SPLs, highlights what makes SPL evolution challenging, and points to previous mapping studies in the SPL field.

2.3.1 A brief on SPLs

SPLs aim to support the development of a whole family of software products through systematic reuse of shared assets [CN01a]. These assets give support to different stages of the SPL production process. The asset list includes variability models (i.e., allowed variants to be exhibited by the SPL products, a.k.a. features), architecture (i.e., high-level description of the main modules involved and their connections), software components, class libraries, code snippets or at a higher description level, models as in model-driven SPLs. It might also include requirement documents, plans, test cases,

process descriptions, product configurations, and trace documents. These assets are handled along two interrelated processes. During Domain Engineering (DE), the scope and variability of the SPL are defined, and reusable assets are developed. During Application Engineering (AE), products are derived using these assets by resolving variability [PBvdL05b]. Hence, variability management is an SPL hallmark. SPL assets can be of different variabilities: *common* assets are present in all products, *variable* assets are present in some products, and *product-specific* assets are local to individual products.

As any other software, SPLs are subject to evolution [DSB05]. Specifically, we conceive evolution as adaptation of the SPL to cope with *changing requirements*. This might happen in two different scenarios:

- during product derivation, new requirements emerge (a.k.a. reactive evolution). These requirements can be accounted for in two different places: within the product realm or within the core-asset realm. The former implies the creation of product specific artifacts. Application engineers can use the core-assets as basis for further development, or they can develop new assets from scratch. Second option is within the core-asset realm. Here, requirements are tackled by domain engineers, and additions can benefit products other than the one generating the change.

- at any time, SPL engineers must be able to anticipate future needs (a.k.a. proactive evolution). This might lead to adapt core-assets in such a way that the SPL is capable of accommodating the needs of product stakeholders in the shortest amount of time.

Previous scenarios involve **SPL changes**. Figure 2.1 depicts the main types of changes along the lines of those proposed in [BP14, Kla08]. Common functionality can be made variable if it should be excluded from some products. Usually, this requires changing the implementation (to make it variable) which then affects all existing products. Conversely, making a variable asset common, influences at least those products that did not contain the asset before. Making a variable asset product-specific, or a product-specific asset generic, requires also to adapt individual products to hold or unhold the asset, respectively.

The bottom line is that SPL assets might be moved along "the variability spectrum": *common, variable* and *product-specific*. Common assets are present in all products, variable assets are present in some products, and product-specific assets are local to individual products. Moving along this spectrum is not straightforward due to SPL specifics, namely:

- **Large number of asset inter-dependencies.** The distinction between DE and AE introduces dependencies between products and the reusable assets used in their production. DE and AE have their own life-cycles and priorities. The urgency in releasing a product, fixing a bug, providing a new product release, or delivering a new feature may vary depending on whether the stakeholder is involved in DE or AE. Nevertheless, both parties need to be in sync to avoid SPL erosion [DSB05].

- **Broad scope**. SPLs aim to build a family of products. Hence, the volume and likelihood of asset coupling is potentially larger than if the focus were on a single product.
- **Large life-span**. SPLs are long-term investments. This lengthy life-span should encourage a more effective control over SPL evolution in order to avoid SPL decay [vGB02].

A final remark. Terminology was particularly elusive in this study. In the SPL literature, the term "evolution" can denote a broad range of concerns: migrating legacy systems into SPLs (e.g.,[LC13]), refactoring (e.g.,[LC13]) or bug-fixing (e.g., [RB08, SLB13]), to name a few. This is not specific of the SPL literature but it has long been recognized for software engineering in general [BR00]. The term "maintenance" tends to be predominantly used to describe activities aiming at preventing software from failing to deliver the intended functionalities. In the same vein, SEBOK defines maintainability as "the probability that a system or system elements can be repaired in a defined environment within a specified period of time" [SEB]. It can be noticed a bias towards the use of the term maintenance in relation with "failure" and "repair". From this perspective, maintenance predominantly aims at preserving functionality. By contrast, we conceive "evolution" not so much as a repairing action, but as an enhancement in the system's capabilities. Here, stakeholders (rather than bugs) tend to be the main triggers of evolution. This distinction is aligned with the way software modifications are classified by Kitchenham et al. [KTvM+99]. Rather than using Swanson's classification of maintenance activities based on intention (i.e., corrective, adaptive, and perfective) [Swa76], Kitchenham et al. propose to categorize the modifications in terms of activities performed: activities to make corrections (i.e., existence of discrepancies between the expected behavior of a system and the actual behavior) *versus* activities to make enhancements (i.e., existence of desires to somehow change the current behavior of the system). For the purpose of this work, we use the term "evolution" to denote these enhancement activities, would these be modifying the scope, the commonality, the variability or the products of an SPL. We then leave out activities such as SPL migration ([BLL08, LC13]), SPL bad-smell detection ([ANS+04, GpKL14, LP07, BGvS10, PPF+14, VFAC14]), SPL refactoring ([ACA08, AGM+06, RB08, STKS12, SLB13]) or SPL bug fixing ([KSLG11, KSL+13]). At adulthood, SPL is exposed to a wider customer base and hence, the pressure for new functionality increases. As pointed out by Singer, "a corrective activity may require only the ability to locate faulty code and make localized changes, whereas an enhancement activity may require a broad understanding of a large part of the product" [Sin98]. Our research questions are headed for assessing the types and coverage of these "enhancement activities".

2.3.2 Related mapping studies

We conducted a Scopus[2] search for mapping studies in SPLs published from 2010. The following search string was used:

[2] http://www.scopus.com/

Ref.	Year	Topic	Research Questions
[MAI12]	2012	Quality attribute	What quality attributes have been proposed for assessing the quality of software product lines?
			What measures have been proposed for assessing the quality of software product lines and how are they used?
[LC13]	2013	Migration	What approaches have been proposed on SPL oriented evolution and what is their focus and origin?
			Which challenges for SPL oriented evolution have been identified?
[LBd+13]	2013	Risk management	Which risk management steps are suggested by the approaches?
			Which risks were identified and reported in SPLs?
			Which risk management activities and practices are adopted by the SPL approaches?
			What do the researchers commonly use to evaluate the identified risks?
			How do the stakeholders influence the identified risks?
[PCF14]	2014	Management tools	How many SPL management tools have been cited in the literature since 2000?
			What are the main characteristics of the tools?
			What are the main functionalities of the tools?
[SdOdA15]	2015	Consistency checking	What kind of consistency checking activities have been performed in the literature?
			Can any trend on consistency checking be recognized in the research field?
			How do the existing approaches relate to each other?
[HPMFA+15]	2015	Bibliometric analysis of SPL research	What are the most influential papers on SPL literature?
			Who are the most prolific authors?
			What journals, conferences, etc. have published the majority of the papers?
			How numerous is the SPL literature? How has paper publication been distributed over time?
			What are the main topics studied in the area? How has the interest in those topics evolved with time?
			What are the most impacting papers for a given topic along a certain period of time?

Table 2.1: Related mapping studies.

("software product line" OR "SPL") AND ("systematic literature review" OR "systematic review" OR "research review" OR "systematic overview" OR "mapping study")

We identified six relevant papers that overlap with our interests (see Table 2.1). For quality attributes in SPLs, Montagud et al. [MAI12] found 165 measures proposed in the literature. This figure is broken down along the SPL life-cycle phase in which the measures are applied: *Requirements* (9%), *Design* (67%), *Realization* (4%), *Testing* (3%), *Application domain phase* (7%), and, most important here, the *Evolution stage* (10%). The latter is based on the insights of a single paper: [AD07].

Laguna et al. [LC13] address the reengineering of legacy systems into SPLs. Here, the term *evolution* is understood as the effect of migrating a set of related products, probably created by clone-and-own operation, into an SPL where reusable assets are obtained by refactoring existing products. Our focus is not so much in how SPLs are created by reengineering existing products, but SPLs' assets evolution. Indeed, studies of Laguna et al. present no overlap with our primary studies. Though refactoring is certainly a trigger for evolution, we are more interested in how SPL engineers accommodate new functionality. This, makes Risk Management (RM) a topic of special interest. The mapping study conducted by Lobato et al. [LBd+13] identifies RM activities and practices in SPLs. Some practices tackle the evolution of SPLs. For instance, the practice *SPL variability* acknowledges that "the product variability must be considered when evolving the architecture". However, SPL evolution does not appear as a first-class activity but is scattered among other steps (e.g., *SPL management*, *SPL variability*, *SPL testing*, etc). By contrast, we move SPL evolution to the forefront, aiming to provide a broader overview of the different aspects involved, not limited to RM. Nevertheless, all the references concerning evolution were also included in our study.

Pereira et al. [PCF14] focus on SPL management tools. A classification facet is about the functionality cluster supported by the tool: *Planning* (i.e. means for collecting the data needed to define domain scope), *Modeling* (i.e. means for represents the domain scope), *Validation* (i.e. means for validating the domain), *Product configuration* (i.e. means for product derivation) and *Import/Export* facilities. The outcome provides the following distribution: *Planning* (34%), *Modeling* (85% of the tools support at least four of the functionalities), *Validation* (49% support at least three of the functionalities), *Product configuration* (83%) and *Import/Export* (71%). However, evolution as such is not explicitly considered but blurred behind other notions, mainly the *Validation* cluster which comprises functions for the inclusion of new requirements. It is not clear the extend to which tools give support to the evolution life-cycle (see later).

For consistency checking, Santos et al. [SdOdA15] undertook a mapping study for 24 primary studies. This work is certainly of interest for SPL evolution. Indeed, consistency checking aims at assuring that all SPL assets remain consistent with each other after some changes have been introduced: *model against source code* (25%), *model against model* (33%), or *model against specifications* (42%), where rates are those provided by this study. Our work extends beyond consistency checking to include other activities of the change life-cycle [YCM93]: identify change, analyze and plan

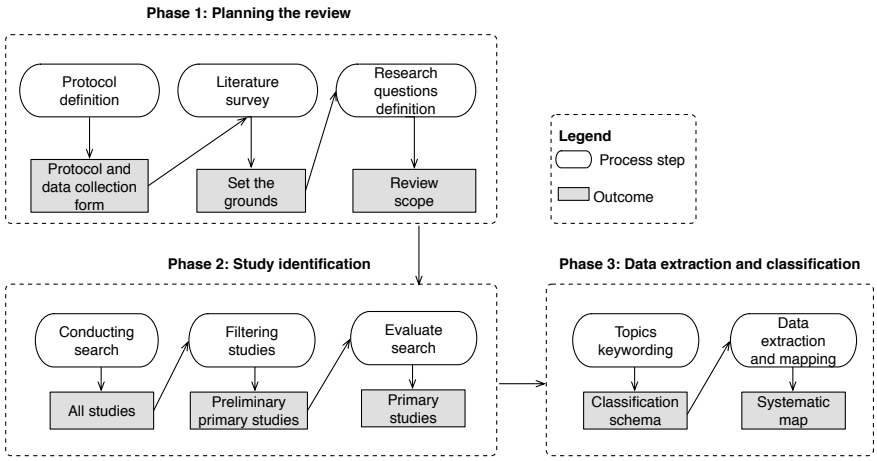

Figure 2.2: Systematic Mapping Study process (adapted from [PFMM08a]).

change, implement change or verify change.

Finally, Heradio et al. [HPMFA+15] perform the broadest mapping study on SPL research. Authors analyzed 20 years of the SPL literature (from 1995 to 2014), which involved above 2800 primary studies. Authors, resort to bibliometric analyses to: (1) identify the most influential publications on the SPL literature (based on received citations), (2) detect the most covered SPL "research topics" (in terms of published papers), and (3), determine how the interest in these research topics evolved over time. Main research topics are: *software architecture*, *automated analysis*, *feature modeling*, *software reuse*, *variability management*, *software quality*, *product derivation*, *domain engineering*, and *software design*. Regarding the evolution over time, authors ascertain that: (1) *software architecture* was the initial motor of research in SPLs; (2) work on *software reuse* has been essential for the development of the SPL research; and (3) *feature modeling* has been the most important topic for the last fifteen years, having the best evolution behavior in terms of number of published papers and received citations. From our perspective, it is worth highlighting that SPL evolution does not appear as a first-class topic, but included as part of *software reuse* and *software design.*

These studies can be considered good sources of information on their subjects. Yet, SPL evolution tends to be blurred behind other notions (e.g. migration, risk management, consistency checking, etc.). We aim at moving SPL evolution at the forefront by providing a deeper analysis along the lines of the change mini-cycle stages [YCM93].

2.4 Method

A Systematic Mapping Study (SMS) is an evidence-based form of secondary study. It provides a wide overview of a research area, to establish if research evidence exists

on a topic, and provides an indication of the quantity of the evidence [KC07]. SMSs offer multiple benefits [BTBK08]. First, SMSs identify gaps and clusters of papers based on frequently occurring themes, using a systematic and objective procedure. Second, SMSs help plan new research, avoiding effort duplication. Third, they identify areas suitable for future systematic literature reviews (SLRs), a more in-depth form of secondary studies with a focus on smaller research areas and more concrete research questions compared to SMSs. The software engineering community is working towards the definition of a standard processes for conducting SMSs. Guidelines and procedures for undertaking SMSs are defined in [BTBK08, PFMM08a, PVK15]. Similar to other studies (e.g., [dCM+11] and [TGAS14]), we split the process proposed by Petersen's. [PFMM08b] into three main phases (see Figure 2.2):

- *planning the review*, where the need for the review, appraisal of related literature surveys and research questions are set. Similar to other SMSs [dCM+11, TGAS14], we complement Petersen's. approach with a protocol definition process and the data collection form as suggested by Kitchenham et al. [KC07],
- *study identification*, where relevant papers are identified. First, a set of initial papers are identified by querying digital databases. Then, these studies are filtered based on inclusion/exclusion criteria, yielding primary studies.
- *data extraction and classification*, where primary studies are analyzed to derive the classification schema, and studies are classified under the schema.

Next subsections provide the details.

2.4.1 Phase 1: Planning the review

This section introduces the directives for planning our SMS, along Kitchenham's guidelines [KC07]. This step iterates along three activities: *protocol definition*, *literature survey* and *research question definition* (see Figure 2.2-"Phase 1"). We analyzed literature surveys on SPL evolution whose outcome is presented in Section 2.3. As for the research questions, we point readers to the introduction, where the objective of research questions RQ1, RQ2, RQ3 and RQ4 is set. Hence, this section focuses on the protocol definition.

2.4.1.1 Protocol definition

This includes the need, the topic and the scope of the review, the preliminary research questions, a preliminary search strategy, selection criteria, and a data extraction form [KC07]. We reviewed and updated the protocol in several iterations throughout the entire SMS process.

The need for the review. This SMS is motivated by the perceived need to systematically map out efforts made on SPL evolution. Thus, the outcomes of this study can identify the trends, hotspots and gaps which need attention from the community. Moreover, leading venues to publish results (and read literature) on SPL evolution can be identified. In addition, researchers and practitioners can check if there

is a growing or decreasing interest on SPL evolution. An overview of the field and its distinctive concerns is given at the beginning of this work (Sections 2.2 and 2.3).

Preliminary research questions. The goal of this study was to obtain a comprehensive overview of current research on SPL evolution.

The search strategy. The search strategy must lead to inclusion of relevant papers and exclusion of irrelevant papers. We set initial search strategy to include querying digital databases with customized search strings, followed by manual filtering of the resulting studies by predefined inclusion and exclusion criteria. To avoid replication, we detail this process later in Section 2.4.2.

Inclusion and exclusion criteria. For filtering the papers, we formulated inclusion and exclusion criteria. The inclusion criteria are:

- IC1. The study focuses on SPLs as opposed to peripherally addressing the topic.
- IC2. The study focuses on SPL evolution as such. Migration from single product to an SPL approach, refactoring, bad-smells and bug-fixing are not considered (as addressed in Section 2.3).
- IC3. The study is peer-reviewed.

Next, the exclusion criteria are:

- EC1. The study is not SPL-centric.
- EC2. The study does not address evolution.
- EC3. The study is in a language other than English.
- EC4. The study is gray literature, extended abstract, tutorial, tool demo, or doctoral symposium paper.
- EC5. The study is a delta of another study in the review.

Data extraction form. Its main purpose is to help researchers in collecting all the information needed to answer the research questions, recording rationales for inclusion and exclusion of the studies, and classifying each of the studies along the classification schema. We employed a spreadsheet to collect metadata for all of the studies: *title*, *authors*, *year of publication*, *publication type, venue*, *abstract*, and *keywords*. Additionally, we gave a brief *summary* for each study and *rationales* for inclusion or exclusion. If a study was included, then we determined its classification *categories*. The resulting table for all primary studies is available at `http://www.onekin.org/content/spl-evolution-mapping`.

2.4.2 Phase 2: Study identification

This phase includes: *conducting the search* and *filtering studies*. Additionally, we added the *evaluating the search* step to verify that we did not miss any important study (see Figure 2.2-"Phase 2"). Figure 2.3 depicts the process.

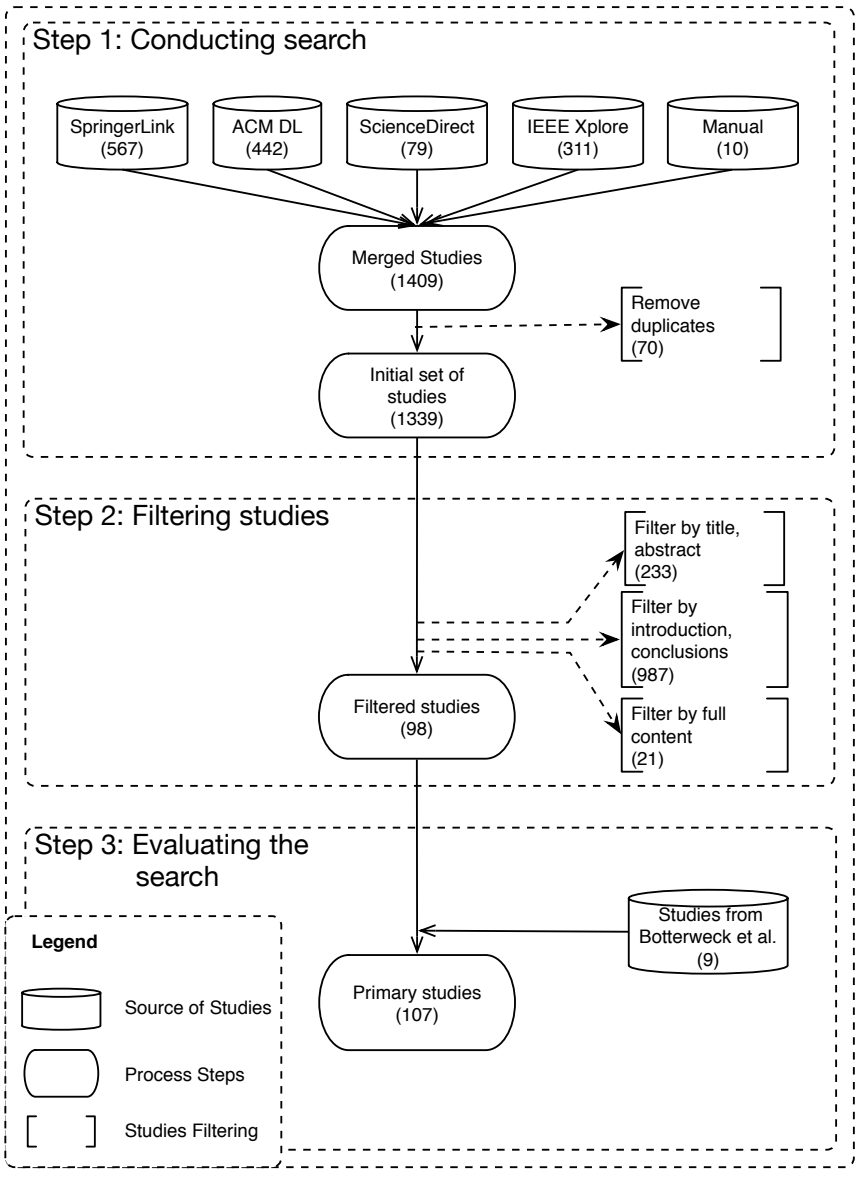

Figure 2.3: Study identification process.

2.4.2.1 Conducting the search

This step deals with building a search string to query digital databases. We followed the PICO approach as suggested by good practices on systematic reviews [PVK15, KC07]. *P* stands for *population*. In our case, population refers to the area on *SPLs*. *I* stands for *intervention*. In our case, the procedure to be assessed is *evolution*. *C* corresponds to *Comparison*. Here, we do not compare different strategies for evolution but assess the area as a whole. Finally, *O* stands for *Outcome* which does not apply to our study either. The identified keywords are then, "Software Product Lines" and "Evolution".

Next, synonyms should be found. Along the guidelines of Petersen's. [PVK15], the following related mapping studies were consulted: [CB11, MMCdA14, LC13, KG09]. Additionally, we conducted a pilot study over the IEEE database to find a balance between *hits* and *noise*. We noticed that the terms "evolution" and "maintenance" tend to be used interchangeably. Hence, we included both terms. This resulted in the following search string:

> *(("product lines" OR "product families" OR "product family" OR "product-lines" OR "product-families" OR "product-family")*
>
> *AND*
>
> *("evolution" OR "evolving" OR "maintenance" OR "maintaining"))*

We restricted the search to studies published up to **July 2015**. The following electronic databases were consulted: IEEE Xplore [3], ACM Digital Library[4], Springer Link[5] and Science Direct[6]. The query was matched against the *title*, the *abstract* and the *keywords*. Unfortunately, at the time of this study, Springer did not account for such focused search, and we resorted to posing the query against the full article content. Additionally, previously known references (identified during the analysis of related literature in the "planning" phase) were manually added. Refer to Figure 2.3 to inspect the number of the studies that each digital database returned. Figure 2.3-"Step 1" highlights how Springer Link returned most primary studies (40,2%). Next, ACM Digital Library, Science Direct , IEEE Xplore, and manually retrieved studies, returned 31,4%, 5,6%, 22,1% and 0,7%, respectively. In summary, we obtained 1409 primary studies in this first step, where 70 were duplicated and hence, removed. This leads to 1339 initial studies.

2.4.2.2 Filtering studies

For filtering, we formulated inclusion and exclusion criteria (already presented in Section 2.4.1.1). A paper was selected as a primary study only when it met all the inclusion criteria and none of the exclusion criteria. Filtering was mainly conducted by one researcher. When the researcher was not sure about including or excluding a paper, the other researcher was asked to discuss and decide. Next, we outline the main debates:

[3] `http://ieeexplore.ieee.org`
[4] `http://dl.acm.org/`
[5] `http://link.springer.com/`
[6] `http://www.sciencedirect.com/`

- EC1 ("The study is not centric to SPL"). Some studies addressed SPLs incidentally, not really focusing on SPLs. For instance, studies just mentioning SPLs as related work (e.g., [AC07]).
- EC2 ("The study does not address evolution"). We found that *evolution* might encompass a great variety of concerns such as migration or refactoring. As noted in Section 2.3, we understand *evolution* as "activities to make enhancements". Hence, we left outside activities such as SPL migration ([BLL08, LC13]), SPL bad-smell detection ([ANS+04, LP07, GpKL14, BGvS10, PPF+14, VFAC14]) or SPL refactoring ([ACA08, AGM+06, RB08, STKS12, SLB13]) or SPL bug-fixing ([KSLG11, KSL+13]). We also excluded studies on traceability with a focus on trace extraction and trace specification ([AKM+10] [MPK12][MCNY07][AC07][SPZ09][VPS+12][YGW12]).
- EC4 ("The study is grey literature"). We excluded grey literature, and also extended abstracts, tutorials, tool demos, and doctoral symposium papers (e.g., [VRG14]).
- EC5 ("The study is a delta of another study in the review"). 26 deltas were excluded in favor of the paper that more extensively detailed the issue (e.g., [tBMP11, BPPK09, WMHB11]).

We applied a three-stage filtering process to the initial set of 1339 studies (see Figure 2.3-"Step 2"). Filter 1 looks at the title and abstract (233 papers left out). Filter 2 looks at the introduction and conclusions (987 papers left out). Finally, filter 3 looks at the content (21 papers left out). At a given stage, a study was filtered out only if the researcher doing the work was fully sure that it met all the exclusion criteria and none of the inclusion criteria. Otherwise, it went to the next filtering stage. If reaching the third stage, the study was revised by the two researchers, and a consensus was reached. The process resulted in 98 primary studies.

2.4.2.3 Evaluating the search

The filtering of studies was mainly conducted by one researcher, which is a threat we were aware of. To reduce the risk of having missed any important study, we followed Petersen et al. [PVK15] guidelines, which recommend to cross-check the resulting studies with a test-set of studies. Our test-set was extracted from the most up to date summary on SPL evolution by Botterweck et al. [BP14]. From the set of Botterweck's references we excluded those that do not met our inclusion/exclusion criteria, and obtained a final test-set of 34 studies. We then cross-checked these 34 studies with our 98 primary studies. The cross-check revealed 9 new references. This rises the number of primary studies to 107.

2.4.3 Phase 3: Data extraction and classification

This phase iterates along two tasks, *relevant topics keywording* and *data extraction and mapping* (see Figure 2.2-"Phase 3").

2.4.3.1 Relevant topic keywording

This process yields the classification schema. Our classification schema includes four facets: "Research type", "Product-derivation approach", "Asset type" and "Evolution activity". The classification schema is grounded in the literature. Specifically, the "relevant topic keywording" process was performed to refine the categories for facet "Evolution activity". We departed from a coarse-grained classification for "Evolution activity" first proposed by Yau et al. [YCM93]. This classification was refined by means of the "relevant topic keywording" process. Within this process, a reviewer read the papers and manually look for keywords and concepts that reflected the contribution of the papers. Afterwards, the set of keywords from the different papers were combined together and clustered to form the fine-grained categories for the "Evolution activity" facet. The resulting fine-gained schema is later presented in Section 2.5, as part of the mapping of primary studies. Next paragraphs provide the description of the four facets.

Facet 1: Research type Description & Derivation Method. The research type reflects the research approach used in the primary study. As other SMSs in software engineering [ER11], research type categories are based on the scheme proposed by Wieringa et al. [WMMR05].

Classification Schema:

- "Experience papers" describe the experience of the authors, usually in practice, using a certain method, technology, etc. Often, these papers are written by people from industry.
- "Conceptual proposals" sketch a new way of looking at existing things, providing a vision or philosophical view on a subject matter.
- "Solution proposals" describe a solution which is usually illustrated with an example, case study, running example, etc. The work is barely or not validated; the proposal is only explained, and it is shown how to apply it.
- "Validation research" describes validation of research that is not deployed in practice, for example, by an experiment, performing some kind of tests, lab studies, etc. Usually it follows a solution proposal. It answers the question: is the proposed solution "good"?
- "Evaluation research" describes an evaluation of research, usually by seeing how the solution works in practice or comparing it with other solutions, pointing out positive and negative points. It is more extensive than validation and often carried out within an industrial setting. It answers the question: is the proposed solution the "right" solution?

This facet somehow serves as an indication of maturity. For instance, the existence of case studies or prototype tools in an academic context indicates at least a certain degree of validation ("Solution proposals" and "Validation research"). On the other hand,

"Experience papers" and "Conceptual proposals" might denote an incipient research area.

This classification schema is disjointed, i.e., a study belongs to a unique category. If a study addresses two categories (e.g., a solution and its validation), the "uppermost" category is selected (e.g. validation). From a maturity perspective, categories rank as follows: "Evaluation research" > "Validation research" > "Solution proposals" > "Conceptual proposals > "Experience papers". Note that both "Validation research" and "Evaluation research" will cover studies that propose *new* solutions (if they are validated or evaluated), as well as papers that address the validation or evaluation of *existing* solutions. Hence, we could not determine whether solutions being evaluated/validated are new or they have already being proposed. For our purposes, this is not an issue since our emphasis is on determining the maturity level of each research area, regardless of whether solutions are new or not.

Facet 2: Product-derivation approach Description & Derivation Method. It refers to the way products are obtained from core-assets. Two approaches are commonly distinguished: annotation-based (a.k.a. negative variability) and composition-based (a.k.a. positive variability) [ABKS13a]. However, if the abstraction level of assets is also considered, a number of studies also address model-driven SPLs. A minority yet practical approach for product derivation is the use of clone-and-own.

Classification Schema:

- "Annotation-based ". Here, the code of all features is merged into a single code base, and annotations spot which code belongs to which feature. *During product derivation*, all code that belongs to deselected features is removed (at compile time) or ignored (at run time) to form the final product [BPSP04, Kru01]. Preprocessors are a case in point. They typically provide facilities for conditional compilation, where marked code fragments in the source code are conditionally removed at compile-time. Annotations are realized through tags, such as *#ifdef* and *#endif*.

- "Composition-based". Here, features are realized as compassable units, ideally one unit per feature. *During product derivation*, all units of all selected features and valid feature combinations are composed to form the final product. Frameworks [JF88], Component-based development, Feature-Oriented Programming (FOP) [BSR03, Pre97], Aspect-Oriented Programming (AOP) [KLM+97] or Delta-Oriented Programming (DOP) [SBB+10] applied to SPLs fall within this category.

- "Model-driven". Here, code is abstracted in terms of models. During *product derivation*, model transformations are used that, ideally, generates the complete product together with all documentation, test cases, etc., in a fully automated way [GS03, VV11]. Model-driven SPLs can follow annotations or composition for variability handling. For our purpose, however, the distinctive aspect is that they abstract the way at which product derivation takes place, let this be "annotation-based" or "composition-based".

- "Clone-based". In early stages of SPL adoption, developers might prefer keeping clone-based generated products separately. Here, *product derivation* is just "clone-and-own". Nevertheless, those products conform a family, where changes in one product might need to be propagated directly to sibling products without the intermediation of an SPL infrastructure [RCC15].
- "Hybrid". This comprises studies that somehow combines or blend some of the aforementioned approaches.

This classification schema is disjointed, i.e. a study belongs to a unique category. Papers addressing model-driven SPLs are so classified, no matter whether annotation or composition is used. In this way, we want to gain a glimpse to the extent model transformation is being involved in product derivation.

Facet 3: Evolution activity Description & Derivation Method. Activities involved in SPL evolution. We tap into the change mini-cycle model of Yau et al.[YCM93].

Classification Schema:

- "Identify change". Customers, product engineers, domain engineers, the target market, maintenance needs or competitors might exert evolutionary forces over an SPL. "Identify change" has to do with monitoring those sources of change.
- "Analyze and plan change". Program comprehension is essential to understand what parts of the software will be affected by a requested change. In addition, the extent or impact of the change needs to be assessed to obtain an estimation of how costly the change will be, as well as the potential risk involved in making the change. This analysis is then used to decide whether it is worth carrying out the change.
- "Implement change". This activity conducts the change. The large number of assets and stakeholders involved in SPLs recommend error prevention and guidance mechanism to be in place.
- "Verify change". Techniques to re-verify the SPL after change are crucial to ensure that the SPL integrity has not been compromised.

This classification schema allows for categories to overlap, i.e. a study might belong to more than one category. A finer-grained schema is later presented in Section 2.5, as part of the mapping of primary studies.

Facet 4: Asset type Description & Derivation Method. Type of the SPL asset being subject to evolution. Types are obtained from the reviewed studies.

Classification Schema:

- "Variability model". Variability modeling is to efficiently describe more than one variant of a system. Different approaches to capture such variability have been proposed: Feature Models (FMs) [Kan90], cardinality-based FMs [KC05], Decision-Oriented Variability Models (DOVMs) [SRG11], and Orthogonal Variability Models (OVMs) [PBvdL05b].

- "SPL architecture". An SPL architecture captures the structure commonalities and structure variability of the SPL products, along the architecture elements: software assets, the externally visible properties of those assets, and the relationships among them [CBT+14].
- "Code assets". Broadly, code assets are the raw material to produce the SPL products. This can range from code snippets to models (in model-driven SPLs). Here, code asset might enclose variability built-in, later resolved during product derivation.
- "Products". Broadly, a product is what is delivered to a customer. Depending on the maturity of the SPL, products might be directly derived from the reusable assets based on feature selection, or rather, require the intervention of product engineers before being ready for release [DSB05].

This classification schema is "overlapped", i.e. a study might address evolution for different assets. Notice however, that studies are classified based on the "evolving artefact", i.e. the artefact that suffers the change first, regardless of whether this change is next propagated to other artefacts. So, a study describing how a change in the variability model percolates to code assets and products, is classified as "Variability model".

2.4.3.2 Data extraction and mapping

Having the classification scheme in place, the primary studies are sorted into the scheme. The classification scheme evolved while doing the data extraction, like adding new categories or merging and splitting existing categories. Data extraction process was performed by one reviewer, who entered data into the data extraction form fields: (i) gave a short description of each paper's contribution, (ii) classified the study into the four facets, and (iii) provided a short rationale why the paper should be in a certain category. The second reviewer checked the outcome of this process and checked its correctness. The outcome of this second review could be *agreement*, *disagreement* or *doubt*. If *disagreement*, the document was read (again) in full appraisal by both researchers, and a consensus was reached. If the classification was still dubious, then the studies' authors were contacted through e-mail. This was the case for 15 papers. Additionally, we contacted authors of other 13 studies, as a cross-check measure. These 28 studies are listed in the acknowledgements to thank the authors for the prompt reply to our request. The mapping of the papers and their brief is provided in Section 2.5. The Appendix holds Table A with the mapping of the primary studies into our classification schema.

2.4.4 Threats to validity

There are several factors that may threaten the validity of systematic mapping outcomes. Main shortcomings include: (i) bias in the selection of studies [BPS+12], and (ii) errors when extracting and classifying studies into detailed categories [ER11]. Additionally, we evaluate this mapping study along Petersen's. evaluation rubric [PVK15].

2.4.4.1 Selection of studies

Biases might happen during both finding and filtering primary studies. The former has to do with coming up with primary studies. Here, one of the risks is the lack of standard languages and terminologies [DD08]. To reduce this risk, we refined the "search string" by (i) consulting the keywords used on related mapping studies, and (ii) conducting a pilot study, which let us determine the "noise" introduced by the selected keywords. Additionally, we referred to the main publishing houses in computing science (i.e., ACM, IEEE, Springer and Science Direct), even knowing that a large overlap could exist (indeed, 70 duplicates were detected). Inclusion and exclusion criteria were established to provide an assessment of how the final set of primary studies was obtained. Where in doubt, the screening of a study went from the abstract, introduction and conclusions, to the full-text appraisal. If after full text appraisal, doubts persisted, then the decision about whether to include or not the study was jointly taken by the two researchers. This was the case for 21 primary studies (see 2.3).

In addition, we follow recommendations by Casteleyn et al. [CGM14] to set aside "delta papers", i.e. papers that provide minor additions compared to previously published work of the authors. Inclusion of delta papers might mislead summarization data, specifically if classification is fine-grained with few studies for each facet. This process led to the identification of 26 delta papers. As a final validation, we conducted a cross-check with the two main potentially overlapping survey studies, i.e. [LC13, BP14]. Specifically, primary studies of Laguna et al. [LC13] present no overlap with our primary studies. As for Botterweck et al. [BP14], though this work is not a mapping study but a survey, their references serve to cross-check our's: 25 overlapping, 9 only in Botterweck, and 73 only in our study. Besides enriching our set with 9 new references, this comparison corroborates the role of our work as a systematic mapping endeavor by introducing 73 new references.

We cannot rule out threats from a *quality* assessment perspective because selected studies were assigned no scores[7]. However, with the aim of increasing the quality of included studies, we defined exclusion criteria to get rid of potentially low level quality studies, such as those excluded by "EC4" (grey literature, extended abstract, tool demo, workshop proposal). Additionally, the selected digital databases (ScienceDirect, ACM, IEEE Xplore, and SpringerLink) which are regarded as reliable by the community. Some systematic reviews that include them are: [DD08, LC13, MMCdA14].

Another threat might be the focus on those studies that specifically target SPLs. We did not explore whether other software engineering studies addressing evolution, could be applicable for SPLs. Moreover, our notion of *evolution* can be regarded as too restrictive as we did not consider SPL migration or SPL refactoring.

2.4.4.2 Classification errors

It is possible for authors to introduce bias during the data extraction process. To reduce this risk, we based the data extraction on the words used in each publication wherever possible. First, an author conducted the data extraction and classification

[7] In SMSs, *quality assessment* is not a mandatory practice [PFMM08a].

process. The outcome of this second review could be *agreement*, *disagreement* or *doubt*. If disagreement, the document was read in full (full appraisal), and a consensus was reached. If the classification was still dubious, then the document's authors were contacted through e-mail. This was the case of 15 papers. As a crosscheck, we additionally contacted authors of 25 papers, although only 13 did finally reply. No inconsistencies were appreciated except for the facet "Research type": 5 authors would classify their paper differently w.r.t to this facet. The main confusion originated from the distinction between "validation" and "evaluation" research. Additionally, some authors misunderstood when a study should be considered an "experience paper". This is not totally unexpected. Wohlin et al. [WRdMSN+13] already pointed out how misleading this facet can be. The authors reveal how two independent studies classified the very same papers differently, w.r.t the "Research type" facet. This blurriness might advice to stick to the classification of a single observer that makes clear his understanding of this facet's values, and where the assessment of which research type was conducted is based uniquely on what it is described in the paper. The alternative would be to collect the answers of the 67 studies' authors whose understanding of what "validation" and "evaluation" is might differ, and whose appreciation might be partially biased from experiences not always fully documented.

2.4.4.3 Evaluation rubric for this mapping study

Petersen et al. [PVK15] devise an evaluation rubric where to assess the quality of a mapping study process. This rubric can be used for readers to quick assess the actions undertaken in a SMS. Specifically, authors identify 26 actions worth applying. The more actions taken, the higher would be the quality of a SMS. Table 2.4.4.3 outlines the actions undertaken in this SMS. Additionally, we include a fourth column which points to the Section in which the action is addressed. According to the findings of Petersen et al., the median quality of the analyzed SMSs is 33%. This SMS undertakes 15 out of the 26 suggested actions, which yields a ratio of 57%.

Phase	Actions	Applied	Refer to ...
Phase 1	Motivate the need and relevance	✔	Introduction & Background & Protocol definition (Sections 2.2 & 2.3 & Section 2.4.1)
	Define objectives and questions	✔	Introduction & Protocol definition (Section 2.4.1)
	Consult with target audience to define questions	•	-
Phase 2	**Choosing search strategy**		
	Snowballing	•	-
	Manual	✔	References from [BP14] (Section 2.4.2.2)
	Conduct database search	✔	ACM, IEEE, SpringerLink & ScienceDirect (Section 2.4.2.1)
	PICO	✔	Phase 2: data collection (Section 2.4.2.1))

	Consult librarians	•	-
	Iteratively try finding more relevant papers	✔	Conduct a pilot study (Section 2.4.2.1)
	Keywords from knows papers	✔	From papers [MMCdA14, CB11, LC13, KG09](Section 2.4.2.1)
	Use standards, encyclopedias, and thesaurus	•	-
	Evaluate the search		
	Test–set of known papers	✔	Test-set references from [BP14] (Section 2.4.2.2)
	Expert evaluates result	•	-
	Search web-pages of key authors	•	-
	Test–retest	•	-
	Inclusion and Exclusion		
	Identify objective criteria for decision	✔	Inclusion and exclusion criteria (Section 2.4.1)
	Add additional reviewer, resolve disagreements between them when needed	•	-
	Decision rules (what to do when doubts)	✔	Postpone paper to next filtering level & ask second reviewer (Section 2.4.2.2)
Phase 3	**Extraction process**		
	Identify objective criteria for decision	✔	Provided along the classification schema (Section 2.4.3.1)
	Obscuring information that could bias	•	-
	Add additional reviewer, resolve disagreements between them when needed	✔	We asked authors of 28 studies (Section 2.4.4.2)
	Test–retest	•	-
	Classification scheme		
	Research type	✔	Facet “Research type” included (Section 2.4.3.1)
	Research method	•	-
	Venue type	✔	Venues and frequencies reported (Figure 2.5)
Validity disc.	Validity discussion/limitations provided	✔	Validity evaluation reported (Section 2.4.4)

Table 2.4.4.3. Actions conducted in this SMS: taken (✔) & not taken (•).

2.5 Mapping of primary studies

This section provides a short summary for the primary studies. This implied a more carefully reading not just of the abstract but the whole content. This permitted a finer-

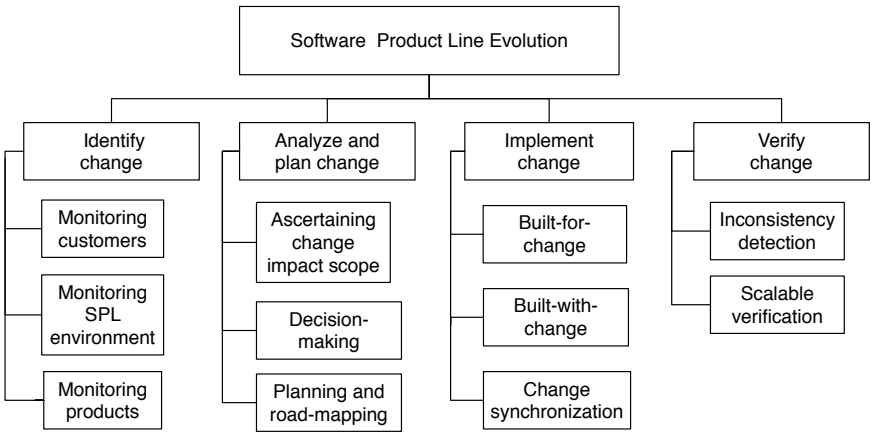

Figure 2.4: Elaborating on the "Evolution activity" facet.

grained elaboration of the facet "Evolution activity" based on the challenged addressed by the primary studies (see Figure 2.4). Table A, in the Appendix A, provides the outcome. Next, we dedicate a subsection to each of these nine activities. For each activity, we first outline what makes this activity challenging for SPLs. Next, we provide a brief about how these challenges are addressed in the primary studies.

2.5.1 Identify change

SPLs broader scope and larger life-span make asset evolution unavoidable. Asset evolution happens in response to forces both outside the SPL organization and within it. By monitoring these forces, engineers can identify emerging needs that the SPL may support. Studies differ in the *force* being monitored: customers, SPL environment, or products (i.e., product engineers).

2.5.1.1 Monitoring customers

Customer needs can be identified trough requirement volatility analyses. Requirement volatility is the tendency of requirements to change over time in response to evolving needs [PYZ11]. In SPLs, requirement volatility tends to be higher due to its broader scope. Here, requirement volatility analysis helps to predict which requirements might change and how. The analysis is based on the priorities that customers assign to each of the SPL requirements. Hence, by monitoring changes to these priorities, engineers identify the set of the requested adaptations, e.g., new requirements may arise, others become obsolete, others may shift from mandatory to optional, etc. This approach is investigated by Savolainen et al. [SK01] and Villela et al. [VDJ10]. An SPL requirement-based taxonomy is provided by Schmid et al. [Kla08].

2.5.1.2 Monitoring the SPL environment

Discussion forums, competitors' websites and market studies can provide useful data silos where to mine future SPL needs. Bockle et al. [Böc05] discusses measures to monitor the SPL environment, including: (1) workshops and discussion forums, (2) usability labs where customers can play with new products and where ideas and complaints are collected, (3) prototypes for new products, and (4) competitors.

2.5.1.3 Monitoring products

Product engineers are responsible for providing feedback to domain engineers. To spur product-engineer feed-backing, Carbon et al. [CKM+08] adapt the agile practice "planning game" [Pla] to SPLs. By means of so-called *reuse stories*, product engineers are instructed to provide concrete suggestions about how to improve the reusability of SPL assets. In addition, product engineers might develop product-specific assets. These assets may "inspire" domain engineers. This is illustrated by Mende et al. [MBKM08] and Creff et al. [CCJM12] where code analysis tools are developed to identify product-specific assets candidate to be promoted as SPL core-assets.

2.5.2 Analyze and plan change

Even to a larger extent than for single products, SPL assets exhibit numerous dependencies: (1) intra-feature dependencies (e.g., <*excludes*> or <*includes*> dependencies in variability models); (2) feature-to-code dependencies (a.k.a. configuration knowledge) or (3), product-to-feature dependencies, which are tracked through product configurations. This coupling makes changes rarely be a one-off event. Hence, *ascertaining the change impact scope* is a first step to decide whether or not to carry out the change. This requires of *decision making* processes tuned to the kind of change being considered. For instance, changing the variability model does not have the same implications than changing a code asset. If the change goes ahead, then *planning and road mapping* come into play. Next, we look into these issues.

2.5.2.1 Ascertaining the change impact scope

Change Impact Analysis (CIA) is defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [Boh96]. CIA scope depends on the asset at hand. Variability models are those with broader impact when evolved. This explains why CIA for variability models has received most attention. But it is by no means the only one. Table 2.3 depicts different change scenarios arranged along the root of the change ("source") and its ripple effects ("target"). Note that it is possible for a study to give support to more than one scenario. Next we provide a paragraph for each row.

A change in the variability model might impact ...

... the variability model itself. Paskevicius et al [PDŠ12] resort to Prolog rules to assess how changes in the Feature Model (FM) affect other parts of the FM. The

Triggering Source \ Triggered Target	Variability model	Architecture	Code asset	Product
Variability model	[PDŠ12], [HVLG12]	[HVLG12]	[Liv11], [HVLG12]	[TBK09], [DKvDP15], [MARC13], [HRGL12], [MW11], [HVLG12],
Architecture	[HVLG12]	[HVLG12], [DPG14]	[HVLG12]	[MW11], [HVLG12]
Code asset	[HVLG12]	[HVLG12]	[YM12], [JZZZ08], [PHS11], [HVLG12] ,[RBK14]	[HVLG12], [MW11]
Product	-	-	[KSS15]	[RCC13, RKBC12]

Table 2.3: CIA scenarios.

FM is expressed in Prolog. For instance, the rule *fm :- all(alt(f1), f2, f3)* describes a FM with *f2*, *f3* as compulsory features, and *f1* as optional. FM changes are also captured as Prolog rules. When the FM is changed, the rule engine computes the set of features affected by the change as a result of the existing feature dependencies (e.g. excludes, includes, and feature associations). The output is the set of features impacted by a change. Heider et al. [HVLG12] present an industrial case study, where they identify engineers' desired trace links when performing CIA in a component-based SPL. Desired traces include links between the variability model and solution space assets (e.g., components, interfaces, and dependencies between them) to ascertain how changes in the variability model impacts the solution space. They further discuss implications for a tool support CIA based on the eclipse IDE.

... code assets. Livengood et al. [Liv11] describe industrial experience on assessing CIA for large and complex variability models (those having multiple constraints). Specifically, authors stress how difficult it is to determine how implementation is affected when variability model constraints are modified. So far, the organization relies on engineers to determine the impact of such changes. Authors advocate for enhanced traceability between the variability model and the code assets.

... product configurations. Changes to the variability model may alter the configuration space (e.g., introducing a new feature adds new product configurations). Thüm et al. [TBK09] present an algorithm to reason about the impact of FM changes on product configurations. The algorithm takes the two versions of the FM (i.e. before and after the change) and classifies changes as follows: (1) *generalization*, if the set of valid product configurations is extended with additional alternatives, (2) *refactoring*, if the same configurations exist, (3) *specialization*, if the set of valid configurations is reduced, and (4) *arbitrary* change, if some of product configurations are removed and others are added. Similar goal but for multi SPLs (i.e., a set of interacting and

interdependent SPLs) is presented by Dintzner et al. [DKvDP15]. Murashkin et al. [MARC13] develop a visual tool to detect the set of product configurations that become non-optimal when the FM changes (w.r.t. quality attributes). In their approach, FMs are annotated with quality values, e.g., *cost* and *usability*. Product configurations are also annotated with expected objectives, e.g., product configuration *p1* can have at most a *cost* of 1500, and *usability* must range between 100 and 300. When a feature quality value evolves (e.g., the cost of a feature increases), the tool highlights those product configurations that do not fulfill the set objectives.

... already derived products. Changes in the variability model may force products to be updated accordingly. Michalik et al. [MW11] propose a preliminary CIA model where to keep track of derived products' configurations, so that whenever the FM changes engineers can assess the affected products. Heider et al. [HRGL12] introduce a tool for domain engineers to get feedback on how changes performed to the variability model may affect existing products. Given a new version of the variability model, the tool re-generates existing products according to their configurations. Next, the tool triggers regression tests for domain engineers to assess the impact of these changes on the re-generated products.

A change in the SPL architecture might impact ...

... the SPL architecture itself. Architectures are the result of design decisions. If those decisions are recorded and contextualized through the features, then so-captured design decision can help to trace core components back to features. This is the insight of Díaz et al. [DPG14]. Consider an ATM SPL. Let's *balanceAccount* be a feature about providing information about user account balance. This feature provides context for the design decision: "*if there is an overload of requests, reject* it". This decision is in turn traced back to the architecture component that implements it (e.g., *Balance* component). On changing feature *balanceAccount* (e.g., adding new variations or excluding dependencies), CIA can go down to the potentially affected components. This scenario gets more complex when design decisions might rest on other design decisions so that their algorithm goes down until all affected components are ascertained. Authors evaluate their approach in an industrial case study on smart grids.

... already derived products. Changes in either the component dependencies or the bindings between these components and the features, may force products to adjust to the new arrangement. Michalik et al. [MW11] propose a preliminary CIA model where to keep track of derived products' configurations. Heider et al. [HVLG12] present an industrial case study, where they identify engineers' desired trace links when performing CIA. Desired CIA include assessing SPL architecture changes on (1) derived products, (2) dependencies with other architectural components and interfaces, and (3) features in the variability model.

A change in code assets might impact ...

... the variability model. Changes to component interfaces and component dependencies often affect variability models [HVLG12]. Heider et al. [HVLG12] present an industrial case study. They identify engineers' desired CIA, including how

code assets changes affect variability models. A model is generated based on those desires and a possible realization in Eclipse is discussed.

... code assets themselves. *Clone&own* is not limited to products. Code assets can also be obtained by cloning existing code assets. In this setting, Jiang et al. [JZZZ08] present an automated technique to identify code asset that need to be changed when a code asset changes. For component-based SPLs, Yazdanshenas et al. [YM12] introduce a fine-grained source code analysis (at code line level) where the impact of component line-grained changes in other components is assessed. For annotation-based SPLs, Ribeiro et al. [RBK14] develop an Eclipse-based tool for annotation-based SPLs. Given a point in code (the one to be changed), this tool identifies the set of additional code changes associated to other features that need to be addressed for the change to be completed. For model-driven SPLs, Pichler et al.[PHS11] and Correa et al. [CdOW11] tackle change impact on meta-models and model transformations. Pichler et al. [PHS11] envisage ten changing scenarios and their respective scopes are analyzed. For instance, a change into a meta-model might ripple through the meta-model itself, model-to-model transformations or model-to-text transformations. Based on the classification for meta-model changes proposed by Gruschko [Gru07], Correa et al. [CdOW11] adapt it for model-driven SPLs. For instance, *Non-breaking changes* (NBC) in SPLs are those changes that do not break consistency and variability rules, and therefore, no product is affected. Authors classify changes in model-driven SPLs (feature changes, meta-model changes and transformation changes) according to this classification, and identify eventual ripple effects.

... already derived products. New enhancements in reusable code assets might impact already derived products. Michalik et al. [MW11] proses a preliminary CIA model that keeps track of the configuration details for each derived products.

A change in a product might impact ...

... code assets. Improvement opportunities can be detected by product engineers. Cossio et al. [KSS15] tackle this scenario. For Version Control Systems (VCSs), development histories can be used to trace products back to the SPL release version from where the product was initially derived. Previous release versions that hold the targeted asset can be detected as well, which, in turn, permits to identify which other SPL products might benefit.

... already derived products. In clone-based SPLs, changes made to one clone might be propagated to other clones. Rubin et al. [RKBC12] propose a model to describe information for managing cloned products. Herein, if a clone changes, then this model could point to other affected features within the clone as well as identify other impacted cloned products. The authors discuss the realization through VCSs. In a later study [RCC13], authors approach is validated through a set of industrial case studies.

2.5.2.2 Decision-making

A change request is not a must-do. Developers should first explore the impact of conducting a change. This very much depends on the kind of change being conducted. This subsection classifies studies based on our understanding of the change type being

Decisions to be made	**Primary Studies**
Make Variable / Make Common	[TB07], [TBC08], [KB12], [DSB09], [NRG08], [RR03], [LDSL07], [APT12], [Sch06a], [SS08], [PYZ11], [CGCS04]
Make Generic/ Make Specific	[HGR10]
Product-local change	[GF13], [GF11], [KR13]
New product	[CGCS04], [HFG+10], [TM14], [MKR94], [SV02]

Table 2.4: Classification of studies based on the decision to be taken.

considered. Change types are those indicated in Figure 2.1. Table 2.4 pigeonholes studies based on these change types. Note that it is possible for a study to give support for more than one change type.

Make variable / make common Here, the issue is about finding the right amount of variability. Too much commonality moves the SPL towards traditional single product engineering. On the other hand, more variability broadens the SPL scope at the expense of more maintenance (and upfront investment). On the search for a compromise, decision-making approaches come in handy, specifically, the *WinWin* model [BBHL94] and the *Question Options Criteria* (QOC) model [MYBM91][8]. Thurimella et al. [TB07] propose a combination of the *EasyWinWin* model (i.e., an adapted version of the *WinWin* model) and the *QOC* model. Specifically, the model includes a *question* (e.g. *"what are the changes that have been requested for feature F1?*"), a set of *options* (e.g., changing variability from mandatory-to-optional, from optional-to-mandatory, to add a new feature or to delete a feature), and finally, some *criteria* (e.g., cost to implement each of these options). In this way, Thurimella et al. [TB07] adapt *QOC* to SPLs. Alternatively, Thurimella et al. [TBC08] and Kumar et al. [KB12] enrich variability models with annotations about feature rationales. This information can later be used to assess *what* and *how* to manage variability. This approach is later evaluated by Kumar et al. [KB13].

In the same vein, Deelstra et al. [DSB09] introduces the variability assessment method COSVAM. COSVAM requires engineers to provide both (1) the SPL's variability model, and (2), the required variability (i.e., the variability necessary to accommodate the change request). The tool detects mismatches between the provided and the required variability. If mismatches arise (i.e., existing product configurations become invalid), the tool suggests the set of adaptations needed to overcome the mismatches. However, estimating the cost of such changes is not always easy. Predictive modeling is a process used in predictive analytics to create a statistical model of future behavior. Schackmann et al. [Sch06a] and Sarang et al. [SS08] advocate to

[8] *QOC* models arrange decision making along four steps. First, define the issues (*questions*). Second, identify available solutions (*options*). Third, define the criteria (e.g., estimates about development efforts, benefits and risks) to rate the available options. Finally, a decision (*option*) is selected on this basis.

create such models from past evolution-driven developments efforts. These models can later be used to estimate costs for the different SPL evolution scenarios (e.g., fix a feature, add a new feature, etc.). At this respect, Peng et al. [PYZ11] assess the profit that a change would imply. The metric is based on the following estimates: (1) the probability that the change will emerge (estimated by analyzing the market and the technological trends), (2) the volume of the change (the number of products affected by the change) and (3), the added customer value for each product (estimated by multiplying the price and the relative value of all the impacted problems identified in change impact analysis).

If the focus is on risks assessment, Riva et al. [RR03] present an industrial case study, where architectural assessment helped to determine if a new feature would put under risk the SPL integrity. Architectural assessments are used to identify defects and shortcomings of the SPL architecture. If the architecture is weak, new features may compromise the integrity of the SPL. Here, SPL managers may postpone the new feature until the architecture is ready to support it. On the other side, new features may alter the functionality of already existing features. Hence, a careful analysis of feature interactions is vital. Liu et al. [LDSL07] focus on the identification and modeling of safety-critical feature interactions to determine whether they may cause a hazard. For component-based SPLs, Annosi et al. [APT12] present an industrial experience on risk management when updating COTSs[9]. The upgrade may surface incompatibilities with other features resulting into unforeseen side effects. Authors build a decision model that considers expert knowledge and dependencies between the SPL architecture elements (i.e. existing components) and the COTS candidates.

Make generic / make specific Here, the issue is about making a variable asset product-specific ("make specific") or a product-specific asset generic ("make generic"). For this matter, Heider et al. [HGR10] resort to a *WinWin* model. Key stakeholder roles are first identified (e.g., salesperson, product engineers, customers, SPL managers), and next, negotiation clusters are set (e.g., development, market, management). For each negotiation cluster, stakeholders describe their individual objectives and expectations as *win* conditions. For instance, project managers might favor cheap and fast development while product engineers prefer to develop with reuse despite introducing additional delays. If all stakeholders concur on a *win* condition, then the condition is turned into an *agreement*. Otherwise, stakeholders identify conflicts, risks, or uncertainties as *issues*. Stakeholders seek *options* to overcome the collected issues and explore tradeoffs as a team. *Options* can then be turned into *agreements* that capture mutually satisfactory solutions.

Product-local change When core-assets are enlarged with a "newcomer", a question arises about which SPL products to be used as a test bed. Karimpour et al. [KR13] tackle this issue. They compute the synergy between the newcomer and distinct products in terms of *value* and *integrity*. The *value* is provided by products' customers, based on how much value will be added to the product if the newcomer is incorporated. The *integrity* computes cohesion, i.e. the degree to which (a product's) features are

[9] COTS are pre-packaged solutions usually acquired to a third-party for a fee.

perceived to be related to the newcomer (e.g., *play* and *pause* features of a video-player systems are more cohesive than *play* and *volume* features). The best product candidate would be the one with maximum *value* and *integrity*. In a similar vein, but now focusing on product architectures, Gámez et al. [GF13] resort to *diff* tools to compute the architectural differences between a product's current configuration and the new configuration that will emerge, should the newcomer be incorporated. The output identifies which components must be added or removed from each product. Managers would then assess the cost for producing the upgraded product versions.

New product SPLs can potentially account for a large number of products based on different feature combinations. However, not all products end up being realized. The cost of a product is not limited to generating the product. Besides the potential pressure for product-local changes, a new product is a new asset to be maintained when the SPL evolves. This begs the question: how to decide the introduction of a new SPL product? Studies resort to simulation models. Simulations involve designing a model of a system and carrying out experiments on it as it progresses through time. Here, the model is the SPL ecosystem, and the experiments are about the impact of introducing the new product. Studies differ in the estimate being considered, e.g. development effort, time-to-market, change resiliency or marketability.

Chen et al. [CGCS04] resort to simulations to estimate the development effort and time-to-market. SPL managers should first create the model, indicating: the number of current SPL products, phases on which the different products are (development, release, waiting for core-assets requested), phases on which core-assets are (in development, released), and the number of developers and their current state (free or under development activities). SPL managers can next simulate the desired change (e.g., introducing a new product). The simulation will tell managers about: the time-to-market for the new product, its development effort, and the additional maintenance effort caused by the change. Effort estimates are traditionally obtained based on previous development efforts. Alternatively, simulation of evolution scenarios can be used. For model-based SPLs, Heider et al. [HFG+10] resort to this approach to measure model maintenance effort.

Minh et al. [TM14] aim to predict products' resiliency. Experts specify the prediction of future evolutions in a feature-like model (called *eFM*). Based on both the *eFM* and the current feature model, authors provide a *configuration survivability analysis* for new product configurations. This analysis measures whether a configuration would still be operational in the presence of forthcoming evolutions.

Murthy et al. [MKR94] tackles marketability. *Product marketability metrics* are proposed to capture customer affordability (willingness to pay) and product quality. Though the study focuses on single applications, the authors argue that these metrics can also be useful to assess whether a new product should enter an SPL. An interesting issue is whether product introduction is a one-off event or rather, it might be better to introduce several products as a single shot. Schmid et al. [SV02] discusses the economical impact of these two scenarios.

2.5.2.3 Planning and road-mapping

The change backlog rarely holds a single petition. Rather, distinct changes are often competing for attention and resources. Harmonious evolution requires roadmaps and release plans that guide the evolution journey.

Road-mapping A project roadmap is a simple presentation of project ambitions and project goals alongside a timeline. The aim is to manage stakeholder expectations, and generate a shared understanding across the teams involved. For SPL evolution, a roadmap provides a global vision of the SPL with features and products to be offered some years from now. Pleuss et al. [PBD+12] and Schubanz et al. [SPP+13] propose the use of FMs to describe roadmaps. Such FMs are called EvoFM, which include "the what" and "the why" of the change. EvoFMs are composed of FM fragments. A fragment gathers related features that are added or removed together during the same evolution step. Dependencies between fragments can also be stablished, just like in an standard FM. Each evolution step can then be represented by a "configuration" of the EvoFM, i.e. a selection of fragments that together make a FM. The evolution of a FM can, hence, be represented by a sequence of EvoFM configurations. Authors visualize this sequence in a matrix-like roadmap. The horizontal dimension represents the time line (year), where each column represents an evolution step. Each cell in the plan represents a configuration decision, i.e. whether a FM fragment is applied in that version or not. Moving from FMs to SPL architectures, van Ommering [vO02] proposes for SPL roadmaps to include both products and components, and most importantly, release dependencies between them. Finally, Savolainen et al. [SK08] report experiences from industrial SPLs and suggests key factors for effectively road-mapping, including e.g., decomposing features into sub-features (to better understand feature inter-dependencies), mapping features to component versions (to understand how features are mapped to code), and prioritizing features based on the value that each product gives to each feature.

Release planning A release plan is a company's current understanding of what features are going into the next release, how many effective developers are deployed on it, and the current status of the development effort (ahead, behind, on-time). It differs from road-mapping in that it signifies that there are a subset of selected requirements to be implement, and there are committed resources to implement such requirements. Release planning provides focus to road-mapping. To know which requirements should be part of the next release, requirements prioritization is conducted. Prioritization can be based on distinct criteria: costs and benefits [NRG08], constraints on available resources to conduct the requirements (e.g., person months until next release) or dependencies between requirements (e.g., one requirement includes/excludes another) [IKH14]. The large set of concerns to be considered leads Taborda [Tab04] to specify release plans as matrixes with different layers. Each layer accounts for different SPL release facets: prioritized product features, allocated requirements for each component, estimated development effort, scheduled dates, test plans cases, and delivered product configuration. The author describes the results of practical trials.

2.5.3 Implement change

CIA strives to identifying the potential consequences of a change. The aim is collecting data to decide whether the change ends up being implemented or not. If the answer is yes, then we move to "Implement change". For classification purposes, we arrange studies addressing this activity along three main issues: (1) how to make SPL assets change resilient ("Built-for-change"), (2) how to accommodate change in a reliable way ("Built-with-change"), and (3), how to ensure consistency when changes are scattered across different assets ("Change synchronization").

2.5.3.1 Built-for-change

Studies strive to anticipate change, and reflect about means to make assets change resilient [LRZJ04]. Resilience very much depends on the SPL architecture and the programming paradigm used to implement code assets.

SPL architecture resilience Studies strive to make the SPL architecture steady through evolution. For planned changes, the wired-in variability of SPL architectures accommodates well. However, unplanned changes might compromise the SPL architecture stability. The question is how to ensure long-term viability of SPL architectures considering that unplanned changes are unavoidable. Although no golden-rules exist, Tischer et al. [TBM+12] and Dikel et al. [DKO+97] present some successful industrial cases. In hindsight, authors propose some guidelines: focusing on simplification (finding a balance between features that are needed for "tomorrow" and features that are needed for "today"), adapting for the future (forecasting market and technology trends that are specific to the SPL architecture), establishing architectural rhythm (fix regular architecture and product releases that help coordinate the actions and expectations of all parties), partnering and broadening relations with stakeholders (e.g., when users want changes to a component, they should negotiate directly with the component owner rather than directly change it themselves), maintaining a clear SPL architecture vision across the company (all parties need to know who is responsible for what), and managing risks and opportunities (e.g., review the architecture with customers and stakeholders, tracking and testing the assumptions underlying customer requirements). Deng et al. [DLS05] discuss several evolution challenges for SPL architectures, and proposes a model-driven approach based on automated domain model transformations. Authors advocate that their approach is flexible enough to accommodate changes to the SPL architecture. Finally, Díaz et al. [DPG14] propose an SPL architecting approach that combines (1) an incremental SPL architecture development based on *scrum sprints*, and (2) a modeling technique to specify the SPL architecture and design decisions that led to each architectural element. Authors evaluate whether their approach enables to maintain SPL architectures' flexibility and integrity upon evolving requirements.

Code asset resilience A number of studies evaluate how variation mechanisms perform as for change resilience. Traditional programming paradigms have been assessed by Svahnberg et al. [SB00] and Sharp et al. [Sha99]. Svahnberg

et al. [SB00] compare inheritance, extensions, parametrization, configuration and generation. Additionally, Sharp et al. [Sha99] discuss object-oriented mechanisms, including inheritance, aggregation, generic programming, and conditional compilation. Departing from traditional programming paradigms, newer approaches have been investigated for SPL realization, namely, Aspect-Oriented Programming (AOP), Feature-Oriented Programming (FOP), and Delta-Oriented Programming (DOP).

AOP supports cross-cuts, i.e., functionality that cannot be cleanly decomposed and tangles/scatters around distinct assets. Tesanovic et al. [Tes07] endorse AOP as a suitable paradigm to face cross-cutting evolution. Dyer et al. [DRC13] compare different AOP interface proposals, namely, open modules, annotation-based point-cuts, explicit join points and quantified-typed events. Figueiredo et al. [FCS+08] evaluate AOP strengths and weaknesses compared to conditional compilation in a set of evolution scenarios. Finally, Abdelmoez et al. [AKEs12] contrast the maintainability effort required during evolution of aspect-oriented SPLs and object-oriented SPLs.

Next, FOP, i.e., a composition-based approach that provides the notion of feature as a construct of the programming language. The idea is to decompose code in terms of features (i.e., feature modules). Object-Oriented Programming (OOP) resorts to subclassing for extending a class C1 with additional functionality in subclass C2. In the same scenario, FOP defines a single class C1 but its definition is split into two assets: the *base* and the *feature* so that C1 is obtained by composing *base* • *feature*. There are not two classes but a single class that is incrementally extended to exhibit a new feature. Countinho et al. [FGFd14] evaluate FOP in several evolution scenarios. Authors conclude that FOP seems to be more effective tackling modularity degeneration, by avoiding feature tangling and scattering in source code, than conditional compilation and design patterns. Cafeo et al. [CDG+12] compare AOP, FOP and conditional compilation. Cardone et al. [CL01] propose *java-layers* (JL), a FOP-like approach for Java, and evaluate JL against Object-Oriented frameworks in terms of flexibility, ease of use, and support for evolution.

Finally, DOP. DOP generalizes FOP by allowing removal of functionality, and hence, brings non-monotonicity to SPLs. In DOP engineers start from a *core module* (containing a valid product configuration), and apply *deltas* to remove, add, and modify features. Schaefer et al. [SBB+10] introduce DOP, and compares it w.r.t. FOP in an SPL evolution scenario.

From the previous studies, it can be concluded that there is not a *one-size-fits-all* approach. Hence, hybrid approaches are suggested. *Aspectual feature modules*, a mix between AOP and FOP, is proposed by Gaia et al. [GFFd14]. Similarly, Loughran et al. [LRZJ04] evaluate *framed aspects*, a mix between AOP and *frames technology* (i.e., a language independent textual pre-processor that creates software modules by using code templates and a specification from the developer). Finally, for component-based SPLs, Tizzei at al. [TDR+11] propose *aspectual-components*, a mix between AOP and components. Authors evaluate to what extent this approach supports the evolution of SPLs compared to object-oriented SPLs.

2.5.3.2 Built-with-change

SPL complexity substantiates the efforts to bring assistance during change implementation. Studies differ in the asset being the subject of change.

Changing the variability model Error prevention can be ameliorated through constraints to be obeyed when conducting the change. Romero et al. [RUQ+13] follow this approach by allowing domain engineers to define authorized changes to the SPL. Such authorized changes are specified in a model (the evolution model). This model is next fed to asset editors so that editions should be compliant with the evolution model (i.e., the constraints). Similarly, Borba et al. [BTG12] and Teixeira et al. [TBG15] propose the use of templates. Templates regulate the SPL evolution so that the behavior of the the original SPL products is preserved

Changing the SPL architecture Guidance to conduct change at architectural level is addressed by Hendrickson et al. [HH07], Knodel et al. [KMNL06] and Garg et al. [GCC+03]. The first two resort to a *diff-like* approach to capture differences between the architecture *as-is* and the architecture *as-it-needs-to-be*. This representation states the architectural elements (components, interfaces and connectors) that need to be added, deleted or modified. This assists engineers in determining the changes to be made. Similarly, Garg et al. [GCC+03] present a tool to visualize different versions of architectural models in terms of components and connectors. When a change is implemented at code level, architecture evaluations can then be used to compare the architectural model with its corresponding implementation at code level. This assists developers in determining whether the changes have been thoroughly completed.

Changing code assets Introducing changes at code level can be error-prone. This is more so for composition-based SPLs where code tends to be scattered across a large number of modules. For example, a module can reference classes, variables, or methods that are defined in another module. Safe composition guarantees that a product synthesized from a composition of modules is type-safe. While it is possible to check individual products by building and then compiling them, this does not scale. In an SPL, there can be thousands of products. It is more desirable to ensure that all legal modules are type-safe without enumerating the entire product line and compiling each product [DCB09]. Schröter et al. [SSTS14] introduce a tool for FOP, which tells engineers (while developing), whether their development is type safe, and hence, no compilation errors will await when composed with other modules. For AOP SPLs, Menkyna et al. [MV09] advocate to create a *change catalog*. Once the type of change is identified (e.g. *Adding Column to Grid*), this catalogue helps to get an idea of its realization through AOP constructs (e.g. *Performing Action After Event*). Authors present this catalog using a Web applications as a case study. Finally, Ribeiro et al. [RBK14] address the ripple effect among code assets in annotation-based SPLs. Based on usage dependencies between code snippets (e.g variables, methods), a tool highlights the impact that changes in the definition of either variables or method signatures, have on other snippets using these elements.

Changing products Customers might request product-specific changes. Product engineers might proceed by developing the bespoken code from scratch. However, the SPL mindset recommends to tap into the available SPL's code assets to look for reuse opportunities. Kakarontzas et al. [KSK08] assist product engineers on this matter by selecting the component that offer better reuse opportunities. Using Test-Driven Development, product engineers might resort to SPL components' test cases for both developing and testing the bespoken code.

For model-driven SPLs, code assets are realized in terms of models, and products are obtained through model transformation. Therefore, product specifics should be handled at the model level. But this is not always possible, and product-specifics end up being added at the code level. The issue is that once models become out of sync, any future re-generation of code overrides manual modifications. To solve this problem, Jarzabek et al. [JT11] propose a flexible model-to-text generator. The idea is to let engineers weave arbitrary manual modifications into the generation process rather than directly modify the generated code.

2.5.3.3 Change synchronization

Change synchronization looks at ways to restore consistency. For classification sake, we distinguish between "inconsistency detection" (addressed in 2.5.4.1) and "change synchronization" (this subsection). The former checks whether SPL assets are kept in a consistent state. The answer is basically "yes" or "no". On the other hand, "change synchronization" takes a step further by restoring consistency. Studies propose restore actions for different SPL assets. Differences stem from the asset being restored.

Scenario: keeping the variability model in sync The variability model can hold a set of dependencies/constraints among its features. It is not enough to detect that some of these dependencies no longer hold. The triggering change should be followed by restore actions such as deleting a feature's children, or removing a cross-tree constraint. Guo et al. [GWTB12] introduce a tool to assess those actions for cardinality-based feature models. Dhungana et al. [DGRN10] introduce a tool to propagate changes between fragments of Decision-Oriented variability models.

However, keeping the variability model in sync is not limited to the variability model itself. It might also impact product configurations, which were set in terms of the variability model which is now being updated. Unlike the previous case, now restore actions are not taken at the time the variability model changes but rather, it is up to product engineers to decide when it is the right moment for products to be upgraded. This decoupling requires of a *variability model change log.* This log records *who* made *what* at *when* to the variability model. Based on this log, product engineers can adapt product configurations at the time that they consider most appropriate. Heider et al. [HRG12] tap on this log to assist product engineers in setting some constraints to be followed when working out the new product release w.r.t. the upgraded variability model. Gámez et al. [GF13] consider this scenario for cardinality-based FMs. Constraint-compliant configurations are obtained which might include new features in order to meet the constraints (e.g., to satisfy a *require* dependency). Barreiros et al. [BM14] face large FMs where the options to restore product configurations might be

very large. Authors introduce an algorithm based on the distance between the original configuration and a potential repaired configuration akin to the upgraded FM. The algorithm suggests those with the minimum distance. Finally, Hwan et al. [KC05] also tackle change propagation but for staged configurations[10].

The variability model might also be impacted by changes conducted down in the SPL infrastructure. In the automotive domain, Holdschick et al. [Hol12] consider how potential changes in the so-called functional model (e.g., deletion of components, optional component becomes mandatory) need to be propagated up to the variability model (e.g., reformulate relations with related features, split features, etc).

Scenario: keeping architectures in sync Usually, product architectures are first derived from the SPL architecture. From then on, both the SPL architecture and the product architectures might evolve independently. Domain engineers can extend the SPL scope and upgrade the SPL architecture accordingly. Likewise, product engineers might be forced to make changes to products architectures to ensure accurate and responsive customer service [CN01a]. Temporary deviations between the SPL and product architectures are allowed, but periodic synchronizations might need to be performed. Notice that the triggering change might come from either the domain realm or the product realm.

If the change originates in the SPL architecture (i.e. the domain realm), then products might benefit from including the new enhancements in the next product release. To know how a product architecture should be updated, architectural traces (i.e., those that trace elements from the SPL architecture to products) become vital to determine what to merge. Michalik et al. [MWB11] seek to abstract the level at which this process is conducted. Although SPLs tend to describe their architecture through a model, this is not always the case for products where the architecture might be hidden within the code assets. This leads Michalik et al. to follow a modernization approach where the product's architecture model is first obtained from the product's code; next, this model is enhanced from the improvements conducted in the SPL architecture model; and finally, the so-obtained enhanced model is mapped back to code. The enhancement stage is conducted by comparing the current product's model and the SPL architecture model. These differences will lead product engineers to manually update products.

If the change originates in the product architecture, domain engineers might consider the change of interest for the entire SPL organization. Again, this process is decoupled, i.e., domain engineers do not consider product changes at the time the change happens, but at a later time in accordance with their roadmap. This begs the question of how engineers cherry-pick the interesting changes from the distinct ones the product suffers from the last milestone. Chen et al. [CCG+03] tap into the product's version. First, domain engineers look at the product's version. Second, two versions are selected that isolate the change of interest. Second, differences are obtained. Third, these differences are accommodated into the SPL architecture through an *ad-hoc* algorithm. Unfortunately, the interesting change does not always correspond

[10] Staged configuration is a process whereby product configurations are arrived at in stages. At each stage some feature choices are made.

to one of the product's version. It might well be the case that interesting changes are scattered across different versions. This might substantiate the effort of Shen et al. [SPZZ10] to detect interesting changes from the differences between the current product architecture (no matter the number of releases it has suffered) and the current SPL architecture. Once differences are worked out, domain engineers pick those of interest, and merge then back to the SPL realm.

Scenario: keeping code assets in sync Previous scenario looks at synchronizing architecture assets. Now, we tackle a similar scenario but for code assets. The difference stems from synchronization to be achieved not just between assets but asset versions. Versions introduce variability in time: the very same asset might be available along different versions. This means that products are derived from asset versions, not just assets. The very same core-asset might be included in different products but at different stages of its life-cycle (i.e. different version numbers). Hence, versioning becomes a main synchronization factor. This moves us to VCSs. VCSs are designed to keep track of who did what and when. Broadly, VCSs support "revisions", i.e. a line of development (a.k.a *baseline* or *trunk*) with branches off of this. The branching model defines the strategy for branching off, and merging back [WS02b]. Studies differ in the kind of product derivation process being addressed: *clone-based* and *composition-based*.

For *clone-based*, each product has its own repository. Several authors argue about the benefits of an integrated platform where cloned variants could be managed. Specifically, both Rubin et al. [RKBC12] and Antkiewicz et al. [AJB+14] propose conceptual operations and discuss VCS implications to manage the synchronization of clones. An industrial experience on managing clone-based SPLs is later conducted by Rubin et al. [RCC13]. Authors conclude that an efficient management of clones relies on not only improving the maintenance of existing clones, but also refactoring clones into an SPL infrastructure. From a technical perspective, McVoy [McV15] introduces new VCS operations suited for BitKeeper, which enables opportunistic reuse and synchronization at component-level. Notice that in clone-based SPLs, propagation takes place at the level of products in the absence of a proper SPL infrastructure.

By contrast, *composition-based* SPLs derive products out of core-assets. In a VCS setting, the SPL comprises: one SPL repository where to keep core-assets, and distinct product repositories where to keep single products. Product repositories are derived from SPL repositories. A *link* between both repositories makes change propagations possible. Thao et al. [TMN08] present a home-made VCS tuned for component-based SPLs. Here, *special* branches inside the SPL repository, keep the SPL repository connected with product repositories. Whenever a product repository is derived, a *special* branch is automatically created in the core-asset repository, aimed for change propagation. Specifically, the *special* branch references the product repository's trunk. This branch works like a mirror: if domain engineers merge changes from the SPL repository main development *trunk* to the the *special* branch, the product repository will automatically get these updates. Anastasopoulos [Ana09] and Dhaliwal [DKZH12] differ from the previous studies in keeping both SPL assets and product assets in the very same repository. For Anastasopoulos [Ana09], the vision is realized

for the Subversion VCS. Engineers can perform activities related to evolution such as creating change requests for a given core-asset, knowing if product assets are in sync with core-assets' latest versions, and propagating changes between core-assets and products. *Diff* operations are used to highlight the differences between core components and product components so that differences can later be merged into a product. However, integrating changes from the core-asset branch into product branches is not always easy. When the core-asset branch holds commits related to more than one change request (e.g, adding a new feature, updating a existing one, etc), developers need to selectively cherry-pick the commits related to the change to be integrated. Commonly, change-request tracking system (e.g., Jira) are used to keep the links between change requests and commits (e.g. a new feature f is implemented in commits *c1, c2* and *c3*). This way, product engineers select the change request they want to integrate into their products, and all the commits related to the change request are merged into the product branch. However, developers need to perform these tasks manually. This is error-prone and time-consuming. Dhaliwal et al. [DKZH12] provide algorithms to identify commit dependencies and create groups of dependent commits that should be integrated together. Authors propose algorithms to automatically determine dependencies among the commits by analyzing dependencies among change requests (in Jira), structural and logical dependencies among source code elements, and the history of developers' working collaborations (in Git).

Scenario: keeping feature mappings in sync Change propagation frequently requires a trace infrastructure to ascertain impacted assets. This infrastructure should also be upgraded. To this end, Seidl et al. [SHA12] introduces *re-tracing operations*, e.g. if *class C* is deleted, so should it be feature mappings that contain *class C*, provided domain engineers approve it. When feature traces are not specified into a separate asset but are embedded into code (i.e., feature annotations), Ji et al. [JBAC15] present nine patterns for co-evolving code assets together with their embedded annotations. Finally, Passos et al. [PGT+13] inspect the Linux kernel evolution history over four years to identify twelve patterns. These patterns cover how variability changes affect both feature-to-code mappings (specified through Makefiles) and source code embedded variability annotations (C files with annotated *ifdef* clauses). For instance, if a new optional feature is added, the pattern instructs engineers to add variability annotations into the source code, as well as to extend Makefiles to include the new feature definition.

2.5.4 Verify change

Once changes are conducted, the SPL needs to be revalidated to ensure that the SPL integrity has not been compromised (e.g. through regression testing). The issues are the specifics brought by the SPL assets and scalability. Rather than repeating all tests for each new release (should this be of the feature model, a core-asset or a product), authors strive to find ways that scale to large SPLs to verify that the changes did not have inadvertent effects.

This subsection aligns with the mapping study on consistency checking presented

by Santos et al. [SdOdA15]. The results are quite similar, though here we include a detailed description of the studies that is missing in Santos' et al.

2.5.4.1 Inconsistency detection

Inconsistency detection checks whether SPL assets are kept in a consistent state. The answer is basically "yes" or "no". Studies differ in the asset being checked.

Inconsistency detection for the variability model Quinton et al. [QPB+14] address consistency for cardinality-based feature models. Authors discuss about common changes and the resulting inconsistencies. A tool supports designers in assessing *the where*, *the why* and *the what* of the inconsistency. For decision-oriented variability models, Vierhauser et al. [VGH+12] build a consistency checking framework where developers are given feedback about the constraints being violated at runtime (between the variability model and the code). However, changes in the feature model percolate down to the SPL, and hence, consistency checking should be extended to other assets, specially, product configurations. For instance, promoting a feature from optional to mandatory turns those configurations that do not included the upgraded feature, inconsistent. Besides product configurations, feature traces (i.e., those that link features to their code realization) are also likely to be affected. Consider a product configuration *p1* with features *f* and *g,* being class *F* and class *G* their code realization, respectively. Now, *f* is extended with an optional child (e.g., feature *h)* together with its corresponding code assets (e.g., class *H*). If class *H* is next inattentively mapped to feature *f* (rather than *h*), then product *p1* will no longer deliver the expected behavior. Study [BTG12] devise tools to check whether the behavior of already existing products configurations is preserved upon feature changes. In the same vein, Borba et al. [TABG15] provide a theory about behavior preservation in SPLs upon feature changes. This study is later extended for multi-product lines (i.e., independently-developed SPLs that are later integrated) [TBG15]. Finally, Jahn et al. [JRG+12] develop a consistency checker to detect inconsistencies for decision-oriented variability models w.r.t the SPL architecture model. When engineers change the code assets (e.g., new components are added), the SPL architecture is automatically updated. The tool raises warnings about any inconsistency between the variability model and the SPL architecture. The tool further suggest the engineer how to resolve such inconsistencies by proposing changes to the variability model (e.g., a new feature should be added).

Inconsistency detection for the SPL architecture Different means are proposed in this item: regression testing, functional tests and architecture evaluations. Regression testing for SPL architectures, checks if new defects are introduced into a previously tested architecture. Neto et al. [ddC+12] apply regression testing in two scenarios: corrective changes and perfective changes. Sales et al. [SC11] resort to JUnit tests to detect violations of design rules during SPL evolution. Alternatively, studies Knodel et al. [KMNL06] and Duszynski et al. [DKL09] use architecture evaluations, i.e., the comparison of an architectural model with its source code counterpart. Possible outputs

include: the architectural element converges (if it exists in both the architecture and the source code), the architectural element diverges (if it is only present in the source code) or the architectural element is absent (if the element is only present in the architecture). Next, architects can interpret the results based on the total numbers of convergences, divergences and absences. Finally, Knodel et al. [KMNL06] illustrates how this output is used to evaluate the SPL architecture consistency between its design and the SPL code assets.

Inconsistency detection for products When new releases for code assets are delivered, existing products might need to be accordingly upgraded. Due to frequent upgrades, products might keep unnecessary assets (a kind of bloatware). This superfluous code may be harmful in safety critical domains, hindering runtime performance and smooth evolution. Demuth et al. [DLHE14] resort to functional tests for ascertaining and eliminating the *bloatware* assets from products, as well as for assuring consistency of products when code assets and variability model evolves.

2.5.4.2 Scalable verification

SPLs might include a large number of assets. Lowering verification efforts has to do with reducing the number of assets that need to be re-verified. Approaches differ based on the verification mechanisms being used: model checking, compositional reasoning and regression testing.

Model checkers automatically verify if a *system* satisfies a given *property*. A *property* can be concerned with safety or liveness of the program, such as the absence of deadlocks, but also product-specific behavior can be checked (e.g., in a coffee machine SPL, check that the total cost of a drink is always less than 2$). The *system* needs to be described in a formal notation (e.g. Petri nets, state-transition diagrams). For large SPLs, Cordy et al. [CCS+12] resort to incremental verification. Here, previous verification results are used to minimize the re-verification effort. Specifically, authors try to determine if new added features are *conservative* or *regulative*. A feature is *conservative* to a product if it adds functionality to the product, without altering its previous functionalities. Alternatively, a feature is *regulative* if it doesn't add new functionality to the product but "adapts" previous functionalities. When the SPL evolves and a new feature f is implemented, knowing that f is *conservative* may drastically reduce the number of new products to verify. For instance, any property violated by an old product p is also violated by the new product p after f is added. Hence, if p is known not to satisfy a property, then there is no need to check p again. The scenario becomes more complex when a blend of both conservative and regulative features are added simultaneously. Theorems are provided to determine which subset of products can be left out for verification when such type of features are introduced. Similarly, static analysis techniques are used by Sabouri et al. [SK14] to determine which features *affect* which *properties* (a feature affects a property if it can make the property valid/invalid). In this way, when the SPL evolves (e.g., a feature is modified that adds/removes program statements), this technique identifies the affected *properties*. Here, there is no need to re-verify the properties that are not *affected* by the statement added/modified.

Berezin et al. [BCC98] introduce so-called "assume guarantee reasoning", a compositional model checking approach that verifies each component separately. It is based on decomposing the system specification into a set of *properties* each of which describes the behavior of a system's subset (i.e., components). Components are annotated through an *assume-guarantee* pair. *Assume* describes the properties for the correct functioning of the component. *Guarantee* denotes properties satisfied by the component provided the *assume* clause is met. A component's *assume* may depend on other component's *guarantee*. This approach is taken by Beek et al. [tBMP12], where the SPL architecture is denoted as a set of components chained by *assume-guarantees*. When a component implementation changes, its *assume-guarantees* may change as well. If stable (the *assume-guarantee* pair did not change), products that reuse the component don't need to be tested again.

Similarly, Rumpe et al. [RRSW] resort to a component compatibility approach, based on pair-wise model checking. If a new component version is compatible with the previous version of the products' component, it could be safely replaced. Finally, commonalities and similarities between products' configurations can be analyzed to additionally narrow the set of products to be tested. The idea is to determine a minimal set of products such that the successful verification of such a small set implies the correctness of the entire SPL. Scheidemann et al. [Sch06b] present an algorithm for this matter.

Regression testing is a type of software testing used to determine whether new problems are the result of software changes (refer to Engstrom et al. [ER10] for an survey for single product regression testing practices). The new twist brought by SPLs is that tests can also be core-asset and hence, subject of reuse. For instance, Lity et al. [LLSG12] use model-based testing in delta-oriented SPLs. When a new product is created, the commonalities with existing product configurations is ascertained, and test assets are automatically derived for the brand new product. In this way, product testing is given a head start.

2.6 Analysis of the results

Though it was not the main driver of this research, we depict distribution of studies over publication venues in Figure 2.5. The International SPL Conference (SPLC) is the prime publication venue for SPL evolution research (28%). In 2005, the SPLC committee decided to merge the SPLC with its European counterpart, the Product Family Engineering (PFE) conference, so they are jointly visualized in the chart. Next in the ranking is the Journal on Information and Software Technology (IST) (8%), the International Conference on Software Engineering (ICSE) (7%), the International Conference on Software Maintenance and Evolution (ICSME) (%4), the Journal of Systems and Software (JSS) (3%), and the ICSE co-located International Workshop on Product Line Approaches in Software Engineering (PLEASE) (4%). The top ten is completed by the International Conference on Software Maintenance and Reengineering (CSMR) (3%), the Journal of Science of Computer Programming (SCP) (3%), the Working Conference on Software Architecture (WICSA) (3%) and the International Workshop on Variability Modeling of Software-Intensive Systems

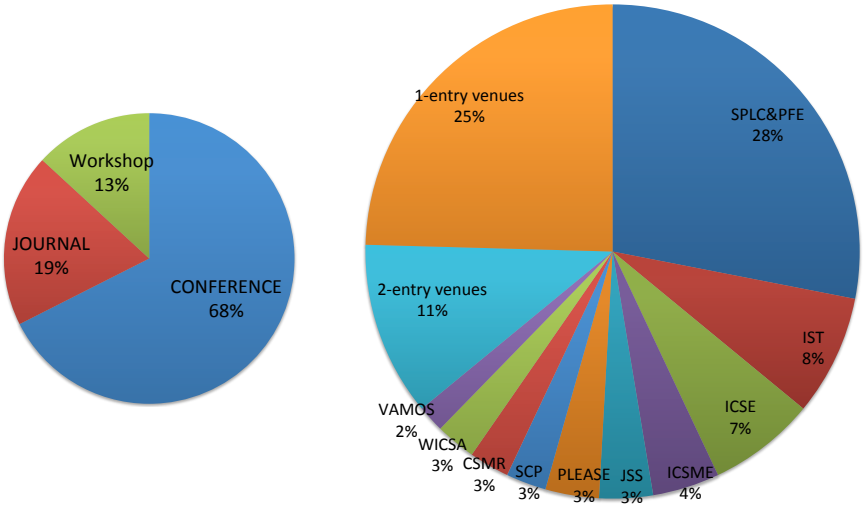

Figure 2.5: Distribution of studies over publication venues: types (left) and individuals (right).

(VaMoS) (2%). The 25% of the publications were unique in the venue they were published in. Figure 2.5 also depicts the type of publication venue. Conference papers and Journals account for the 68% and 19%, respectively, while workshops account for a 13%. These results align with the state of-the-art on SPL evolution by Botterweck et al. [BP14]. Specifically, the majority of the included papers by Botterweck et al. belong to the SPLC (together with the ICSE). Additionally, we both agree on the low numbers of both the International Conference on Software Reuse (ICSR) and the Generative Programming: Concepts & Experience (GPCE) conference. Next, we address each of the research questions.

2.6.1 RQ1: What types of research have been reported, to what extent, and how is coverage evolving?

From the accumulated results shown in Figure 2.6, we observe that "Solution proposals" (31%) is the most addressed category, followed by "Validation research" (24%). As it can be observed, "Solution proposals" have been gradually increasing over the years. "Evaluation research" accounts for a 19%, which indicate the maturity level of the SPL evolution field. Specifically, "Evaluation research" has been lately more increasingly conducted (from 2008 on). This might indicate the SPL field becoming more mature within an industrial setting. Additionally, "Validation research" (24%) studies conducted in academia still need to find their way to industry. "Experience research" (%17) indicates the commitment degree of industry to report "know how", "open issues" and "challenges behind". A few conceptual works have also been addressed (9%), which might indicate incipient challenges being addressed by the

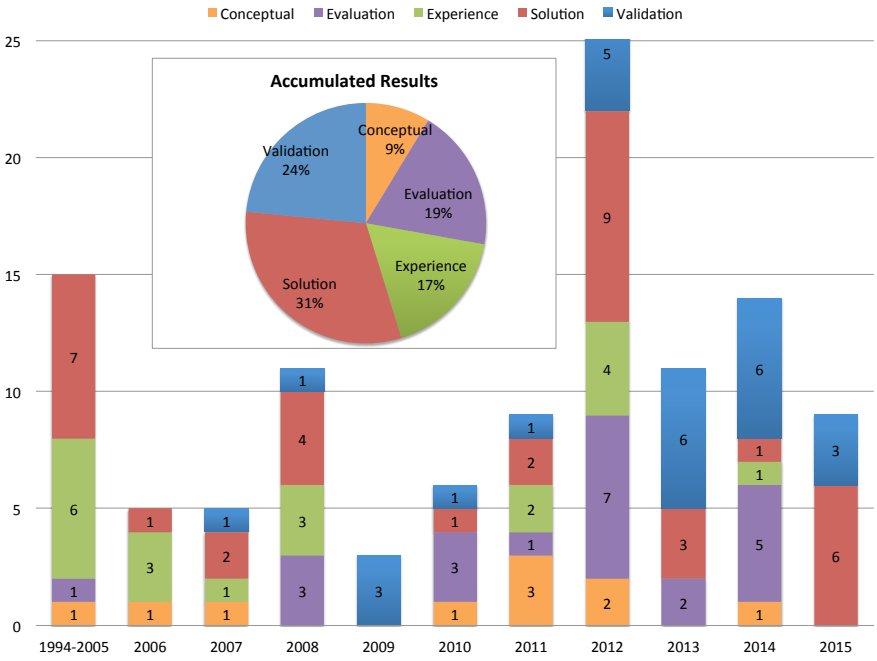

Figure 2.6: "Research type" over time.

community.

From the stacked bar chart, we see a peak of contributions reached in 2012[11]. This peak aligns with other SPL related systematic reviews, which have also identified a global maximum in 2012 [SdOdA15, TAK+14]. Santos et al. [SdOdA15] found also a global maximum with 7 studies (the 29%), while the rest of the years had less than the half of the studies found during 2012, except for 2010 (with 6 papers). Thüm et al. [TAK+14] also identify a global maximum in 2012, with 27 papers. Regarding the evolution of the research, it comes as no surprise that during the fist years (up to 2005) "Experience research" and "Solution proposals" are the ones most addressed. From then on, we observe a trend towards "Validation proposals" and "Evaluation research". Nevertheless, "Solution proposals" still are prominently addressed, which seems to indicate the existence of SPL evolution challenges left to be accomplished.

2.6.2 RQ2: Which product-derivation approach received most coverage, and how is this coverage evolving?

We are interested in assessing how the distinct product derivation approaches are catching on (see Figure 2.7). These approaches might, for instance, impact change

[11] Notice that the survey stops at July 2015. One could postulate that a similar number of papers could be published in the second semester of 2015.

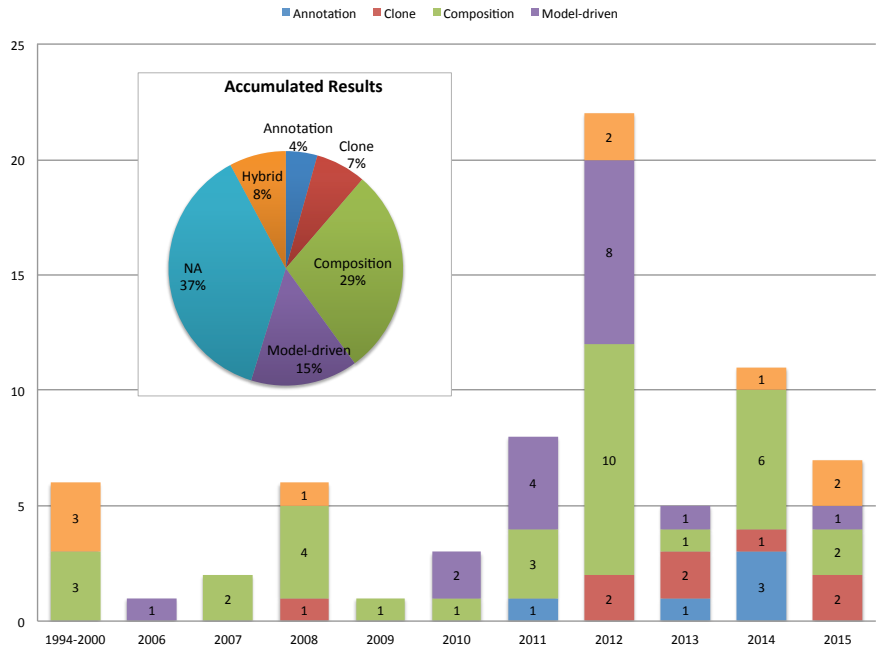

Figure 2.7: "Product derivation approach" over time.

implementation in so far as the structure and code assets might take different shapes tuned for the variability implementation and product derivation approach at hand. This in turn might affect how other activities are conducted from implementing to propagating change. The 37% of the studies are not reporting any specific product derivation approach[12]. The rest of the studies consider either "Annotation" (4%) (e.g., *#ifdef* clauses), "Composition" (e.g., component-based approaches, AOP) (29%), "Model-driven" (15%), "Clone" (7%) or "Hybrid" (8%) approaches. From the stacked bar chart, we can observe how the most addressed one is composition-based, with a share of 29%. This is at odds with the annotation approach being the most widely reported in industry [GLA+09, JB09, PO97, TSSPL09]. This can be due to composition approaches being proposed to overcome the difficulties that annotation-based approaches face when evolved in the large [EBN02, Fav97, KS94]. Interestingly, we can observe an incipient interest on both "Annotation" and "Clone" approaches since 2012 with a share of 4% and 7%, respectively. Although they have been criticized due to its lack of modularity, these approaches have being the subject of recent efforts to overcome this limitation.

[12]This includes studies on external forces (for "Identify change"), variability-model analyses, metrics and negotiation processes (for "Analysis and plan change"), and change synchronization outside code assets (for "Implement change") and inconsistency checking of variability models (for "Verify change").

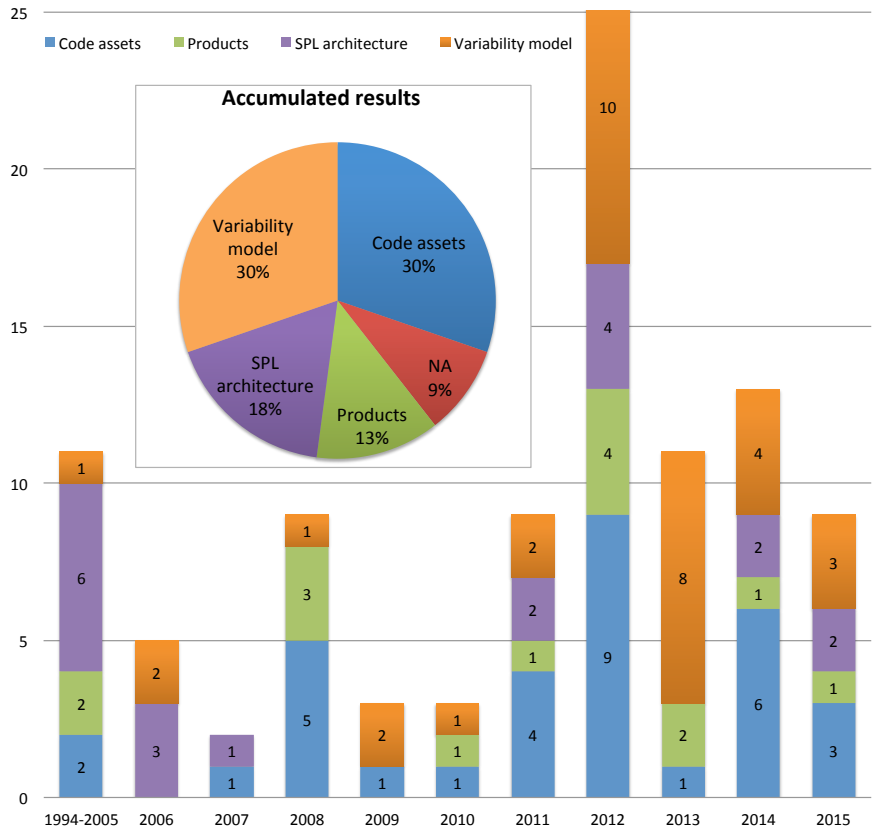

Figure 2.8: "Asset type" over time.

2.6.3 RQ3: Which kind of SPL asset received more attention and how is this attention evolving?

From the accumulated results in Figure 2.8[13], we notice that both the variability model (30%), and the code assets (30%) are the artefacts most addressed. This stems from the way we classified studies. Although studies might deal with distinct SPL assets (e.g., feature-to-code mappings, test assets, etc), here we are interested in the assets that first evolve ("the subject of evolution"), rather than those assets that evolve as a result of the evolution of other assets. The latter assets are not computed into this facet.

"Code assets" account for 30%. Note that this category also includes *models* as the code counterpart in model-driven SPLs. Regarding the evolution over time, "SPL architecture" received more attention during the first years. This aligns with the

[13]"NA" (9%) refer to studies that consider no asset (e.g., a requirement prioritization algorithm [IKH14], monitoring the SPL environment to identify new needs [Böc05], etc).

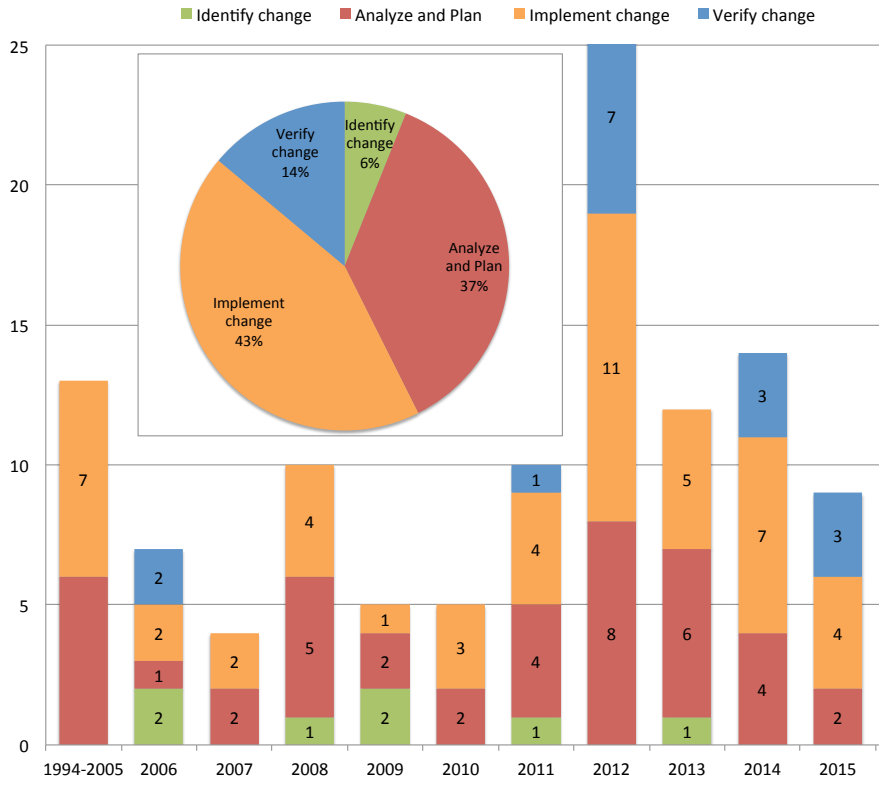

Figure 2.9: "Evolution activity" over time.

findings of Heradio et al. [HPMFA+15]. On the other hand, products lag behind other assets as for attention received (13%). Though some proponents regard products to be derived on the fly from core-assets, the current state of affairs is that products are still in need of being customized, and hence, having a detached life-cycle from the SPL.

2.6.4 RQ4: Which activities of the evolution life-cycle received most coverage and how is this coverage evolving?

Figure 2.9 depicts the rate for each evolution activity. Note that it is possible for a paper to be categorized into more than one activity. This happens in eight cases which explains why the total amounts goes up to 115. From the accumulated results, we observe that "Implement change" (43%) and "Analyze and plan change" (37%) account for more than half of the studies. Conversely, "Identify change" and "Verify change" lag behind with a rate of 6% and 14%, respectively. These differences might be partially explained by SPL challenges being more related to analysis and implementation, while change identification in SPLs exhibits some resemblance with

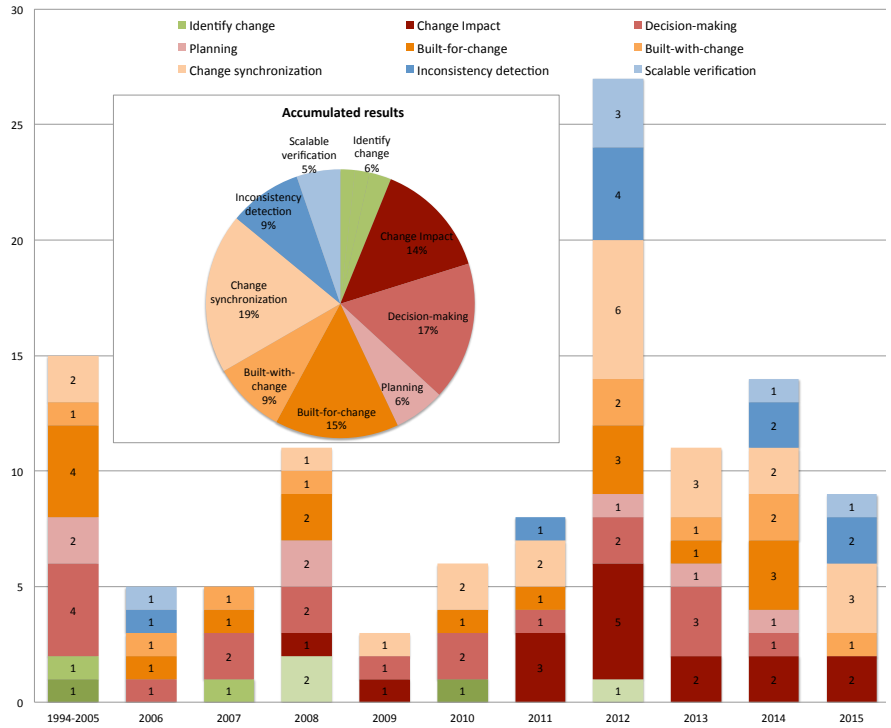

Figure 2.10: A finer-grained classification for SPL "Evolution activity".

single-product engineering. The stacked bar chart shows a sustained interest in "Implement change" and "Analyze and Plan change" over the years, while "Verify change" has recently received more attention.

So far, activities are those of Yau's change mini-cycle [YCM93]. This mini-cycle applies to any software artefact. However, we wanted to zoom into the specific sub-activities SPL practitioners cared about. Based on the mapping of primary studies conducted in Section 4, we refined Yau's model along nine sub-activities (see Figure 2.4). Next subsections provide a finer-grained analysis of those sub-activities.

2.6.4.1 Zooming into identify change

Figure 2.10 highlights this activity as being the less addressed: seven studies. Among the different forces of change, product engineering is the force more broadly addressed [CKM+08, MBKM08, CCJM12], including customers' changing needs [SK01, VDJ10]. This might be so, due to the fact of SPL products being amenable to be promoted as core-assets, a distinctive aspect not applicable to single systems. On the other hand, the forces of change exerted by domain engineers are not so different from those found in single systems, hence, introducing less novelty. This likeness might

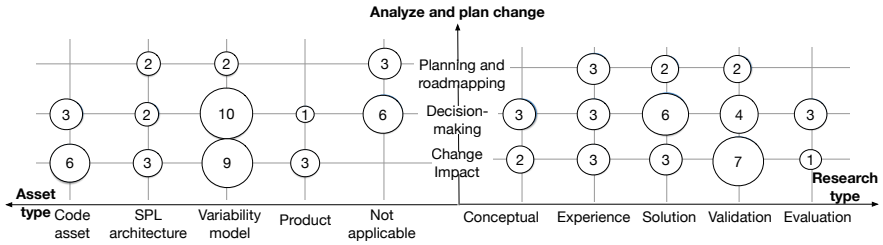

Figure 2.11: Mapping "Analyze and plan change" across facets "Asset type" and "Research type".

also explain the sole existence of a study looking into "the SPL environment" (i.e. competitors, market research and technology forecasts) as a driver for SPL evolution: [Böc05].

2.6.4.2 Zooming into analyze and plan change

Figure 2.10 depicts how "Decision-making" (17%) has received more attention that its siblings "Change impact" (14%) and "Planning" (6%). This might stem from SPLs bringing a new range of decisions concerning how assets evolve along the reuse spectrum. For these sub-activities, we are interested in finding what is the focus (i.e. facet "Asset type") and maturity (i.e. facet "Research type"). To this end, we crossed the activity dimension with these two facets. Figure 2.11 depicts the outcome.

Impact analysis. Maturity level of CIA reveals that proposed techniques are mostly validated within academic case studies or experiments conducted in labs. These studies have mainly considered "Code Assets" and "Variability models" as the evolving assets. Products lag behind. This might evidence that academia barely considers product-specific changes which is at odds with common practice in industry [RDG+07].

Decision making. At first glance, figures suggest this to be a rather mature area with three studies reaching the evaluation stage. However, this first impression should be contrasted against the kind of artefact being addressed. "Variability model" is the most tackled asset with nine studies. This might well stem from the formality brought by variability models that facilitates formal reasoning. However, other assets are largely overlooked. Specifically, the decision about product specifics being promoted to SPL core-assets, has not received so much coverage despite being common in industry [RDG+07]. This is an area that presumably will receive more attention in the future, specially if clone-based SPLs take off.

Planning and road-mapping. Studies seem to rely on industrial experiences to find evidences about how companies schedule and plan releases for SPLs. Variability models and the SPL architecture are the chosen artefacts for this kind of studies.

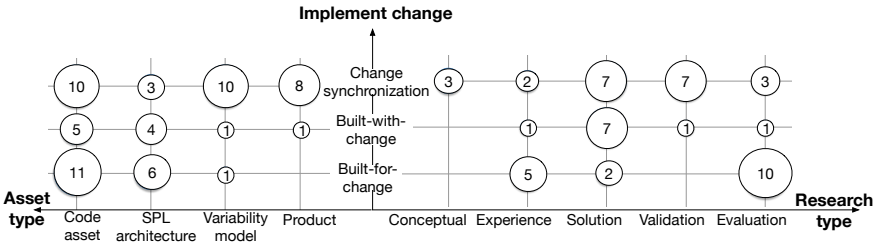

Figure 2.12: Mapping "Implement change" across facets "Asset type" and "Research type".

2.6.4.3 Zooming into implement change

Figure 2.10 places "Change synchronization" (19%) as the most covered activity, ahead of "Built-for-change" (14%) and "Built-with-change" (9%). Next, we analyze each sub-activity w.r.t. asset focus and maturity (see Figure 2.12).

Built-for-change. It comes as no surprise that code-artefact realization is by far the largest studied asset. It also stands out the comprehensive extent at which these studies have been conducted with nine studies reaching the evaluation stage.

Built-with-change. This sub-activity seems to mainly rely on "Solution proposals", and lacks empirical evaluation. Additionally, proposed approaches mostly aid engineers on performing changes at architecture and code asset level. Research on this field seems to underestimate product engineers when conducting product-specific changes (one study).

Change synchronization. This topic is receiving a steady interest in the last years. Special attention is devoted to keeping the SPL assets in sync along all abstraction levels, as well as, to keep synchronized SPL core-assets and product assets. Specifically, "Evaluation research" has focused on keeping the variability model consistent with (smaller parts of) itself [GWTB12], and keeping in sync core-assets and products [DGRN10, HRG12]. The latter calls for effective configuration management approaches. We found several evidences at technical level, i.e., VCSs. For code assets, the trend seems to be to adapt new generation VCSs (e.g., BitKeeper, Git) to SPL's. However, we found neither experiences nor practices regarding how configuration management is achieved in industry.

2.6.4.4 Zooming into verify change

Figure 2.10 gives a rough total for the sub-activities "Inconsistency detection" and "Scalable verification" of 9% and 5%, respectively. Mapping with the other dimensions indicates an evenly distribution of the studies w.r.t. both asset type and research type (see Figure 2.13).

Inconsistency detection. Regarding the asset type, the variability model is the most addressed, presumably due to its readiness to formal reasoning. Specifically, Feature models are the favorite notation as opposed to Orthogonal Variability models,

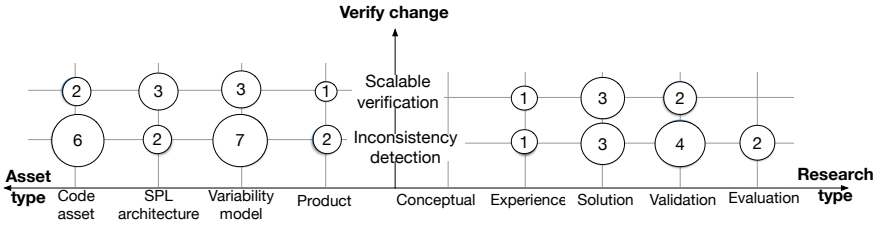

Figure 2.13: Mapping "Verify change" across facets "Asset type" and "Research type".

Decision-Oriented Variability models, or Cardinality-based models. Moreover, more than half of the studies include either validation or evaluation.

Scalable verification. Model checking is by far the most reported approach, and approaches to reduce re-verification effort upon changes, specially, on variability models and SPL architectures. Research in this field looks to be less mature compared to its sibling "inconsistency detection". This might be due to the difficulties in finding industrial cases where to test out the approaches.

2.7 Summary of the results

This Chapter presented the results of a mapping study on SPL evolution. In total, 107 articles were included in this mapping study from 1994 to mid 2015. The aims were (1) to provide a consolidated overview on "SPL evolution", and (2), to identify well-established topics, trends and open research issues. As for the first goal, we described the SPL specifics and their impact on the traditional software change mini-cycle proposed by Yau et al. [YCM93]. On these grounds, we further elaborated on this mini-cycle, and classified the literature accordingly. This permitted a finer grained classification of studies. The answers to the research questions of our mapping study are summarized below.

RQ1, Research type. Solution papers are the most common type of contribution (31%), followed by "Validation research" (24%). Nevertheless, a tendency can be observed towards more evaluation and validation papers. The area reaches a peak in 2012 with 25 papers, and it maintains a steady contribution of around 10 papers a year. Four main conferences stand out as the main venues, though SPLC takes the lion's share with a 28%. Surprisingly, the number of "Experience papers" is rather limited (17%) which contrasts with the increasing use of SPLs in industry [Sav14]. A plea is then for practitioners to report their SPL evolution efforts, rather than reporting only SPL adoption effort. This would certainly be a spur for the whole field.

RQ2, Product derivation approach. Efforts go as follows: "Annotation" (4%), "Clone" (7%), "Hybrid" (8%), "Model-driven" (15%), and "Composition-based" (29%), the later specially for component-based SPLs. Studies on FOP, AOP or DOP took the form of academic evaluations aiming at proving their resiliency upon SPL evolution. No evidences were found on the applicability of these approaches in an

industrial setting. Interestingly, we observed a recent interest in both "Annotation" and "Clone" approaches since 2012 on. Since, both annotation and clone-based SPLs are the approaches widely used in industry, this interest might be interpreted as the research community making the effort to provide means for SPLs in industry.

RQ3, Asset type. Basically, all assets received coverage: variability model (30%), SPL architecture (18%), code assets (30%) and SPL products (13%). Products lag behind other assets as for attention received. This is bad news for SPLs evolving following a grow-and-prune model. Though some proponents regard products to be derived on the fly from core-assets, the current state of affairs is that products are still in need of being customized, and hence, having their detached life-cycle from the SPL. This advices for products to be kept in the radar of SPL evolution.

RQ4, Evolution Activity. The least addressed topic is "Identify change" (%6), followed by "Verify change" (14%). On the other hand, "Analyze and plan change" and "Implement change" have received significantly more attention (37% and 43%, respectively). A finer-grained analysis uncovered some tasks as being underexposed, namely, (1) decision-making on whether product specifics should be promoted to SPL core-assets; (2) change impact analysis upon architectural changes; (3) inconsistency detection for assets other than variability models. "Document change" was left out since no study was found on this activity.

2.8 Conclusion

From the results of this systematic mapping, we can observe that SPLs have received considerable attention by the Software Engineering community, with conferences fully dedicated to this topic. The increasing focus on evolution might be a symptom of maturity where SPL solutions start being tested out. Nevertheless, we have spotted how the SPL research community has left products behind when considering SPL evolution. This means that little support is given for incrementally evolving SPLs from product development. This is unfortunate since capitalizing on the changes that happen at the product level becomes vital during the initial phases of the SPL life-time. As shown in Chapter 1, at these initial phases, commonly the SPL core-asset base does not fulfill all the requirements needed by products, and hence, products need to customize the core-assets, as well as, create brand new assets in order to develop the "remaining" requirements themselves. In this context where both core-assets and products need to co-evolve, the SPL evolution is governed by *pruning* seasons, where product functionalities deemed useful are promoted to core-assets. Specifically, means are required to help SPL engineers:

- identify and analyze how products have changed the core-assets after they were derived from the SPL core-asset base,
- propagate product customizations to the SPL core-asset base, and vice versa.

This thesis aims at addressing both gaps. The next two chapters delve into each of the issues.

Chapter 3

Analyzing product customization

3.1 Overview

The previous Chapter presented a mapping study on SPL evolution research. We saw how the existing research does not sufficiently address the issue of co-evolving both core-assets and products. In such SPLs, evolution is driven by new functionalities implemented in products, and these need to be identified and analyzed in order to elucidate which ones to promote.

In this Chapter[1], we explore how to aid SPL engineers on "customization analysis", i.e. analyzing how products have changed the core-assets they were derived from. Customization analysis is intended to help SPL engineers identify interesting customizations to be promoted to reusable core-assets for the next core-asset release. Deciding when and what should go into the next SPL release is far from trivial. A main decision-making input is the effort that has been put into product customization. We propose the use of data warehouses to analyze this customization effort. Requirement analysis, dimensional modeling and reporting tools are discussed in Sections 3.5, 3.6, and 3.7, respectively. As a proof-of-concept we developed *CustomDIFF*, a data warehouse tool that uses Git as the operational system and pure::variants as the SPL framework. An 8-minute video showing *CustomDIFF* highlights is available at https://tinyurl.com/ycjhwzpc. This research has been motivated and validated in the context of the Danfoss Drives, a SPLC-awarded hall-of-fame company [Dan].

Next Section provides the problem definition.

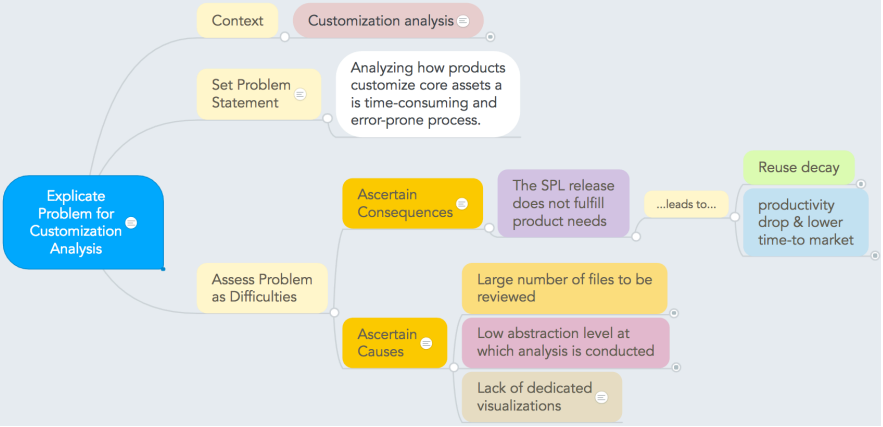

Figure 3.1: Depicting the problem definition for customization analysis with a mind map. Interact with it online at https://tinyurl.com/yay46us8.

3.2 Problem definition

Development in application engineering should be avoided as far as possible, as increases the complexity of SPL evolution management [Kru06]. Unfortunately, companies cannot always avoid it. Different reasons why products need to be adapted after being derived from the SPL release have been reported by the industry: to meet changing products' deadline & budget [DSB05, Jen07, Sch06a], to expedite bug-fixes when close to a release [FSK+16], to speed up unexpected functional changes in customers needs [NNK16, CKM+08, IMY+16], to decrease reusable asset complexity with single-product needs [DSB05, KH12, BB11], and specially, during the first stages of an SPL, where an initial partial SPL does not provide the 100% functionality required by the products, application engineering teams need to develop the "remaining" functionalities themselves [JB09, KST+14, TFC+09]. We refer to this development as **"product customization"**.

Following the *grow-and-prune* model, product customization (i.e the growth) needs to be cleaned up by merging and refactoring (i.e. pruning) [FV03]. The pruning requires SPL engineers to analyse how core-assets are being customized, i.e. looking at the difference between core-assets and namesake assets once customized by products. In this context, a new range of concerns arise: how much effort are product developers spending on product customization?; how and which customizations need to be promoted to the core-asset base?; which are the most customized core-assets?; to which extent is the core-asset code being reused for a given product?; etc. We refer to this endeavor as **"customization analysis"**. Customization analysis is intended to help engineers plan the next SPL release according to products' needs. Evidences from industry revealed that customization analysis is *periodically* performed by *domain*

[1]The content of this Chapter has partially been previously published in [MDA17], and it is currently under revision process in the Special issue on SPL engineering of the Journal of Systems and Software (JSS).

experts, which inspect the *source code versions* looking for any functionality deemed useful. Below are two excerpts from two different industrial case studies:

> *"You must carry out such an effort with the support of the best domain experts of the system.* ***Domain experts*** *are required because only they* ***understand the subtle differences between code unit versions*** *and the needs of the users as they evolved historically, so are best equipped to prune and consolidate". [FV03]*

> *"... all required changes during product derivation are handled through product specific adaptation.* ***Periodically****, the* ***functionality that is deemed useful*** *for the product family is incorporated in the family assets." [DSB05]*

Traditional DIFF utilities might help to see the differences between the core file and the same file once customized by a product [SSRS16]. However, this one-diff-at-a-time approach can hardly scale up to SPLs, where both products and core assets can easily account for hundreds of files. Needed are mechanisms that move from code-level DIFF to assessing differences at higher abstraction terms: features and products. Rather than *DIFF(aFile, aFile)*, we long for *DIFF(aFeature, aProduct)* utilities that encapsulate the scanning of potentially hundreds of products for all the files a given feature has an impact upon. This involves gathering data from thousands of DIFFs. But this is just raw data that needs to be cleaned-up and aggregated in meaningful analysis terms. Due to this issue the following problem arises: **analyzing how products customized core-assets is time-consuming and error-prone.**

Refer to Figure 3.1, which depicts the problem definition as a mind map, and outlines the causes and consequences of the problem. Refer to Chapter 1 for a detailed description on the root-cause analysis of the problem (i.e. cause and consequences of the problem). The reader is encouraged to interact with the mind map at https://tinyurl.com/yay46us8. The nodes can be unfolded to uncover the supporting evidences for each of the claims.

Fortunately, mechanisms already exist that help: data warehouses. Data warehouse (DW) is a collection of decision support technologies, aimed at enabling knowledge workers to make better and faster decisions [KR02].

In this Chapter we study the use of DW for customization analysis. Specifically, our work elaborates around three main research questions:

- RQ1: Which are the information needs for customization analysis? How much time is needed to get these information needs?
- RQ2: To what extent can previous information needs be satisfied through a data warehouse? If so, what would its Star Schema look like?
- RQ3: How can customization analysis be visualized?

This work aims at contributing to the previous research questions as follows:

- RQ1. We introduce a set of questions that might arise during "feature evolution" and "product evolution". The importance and required time to answer to these

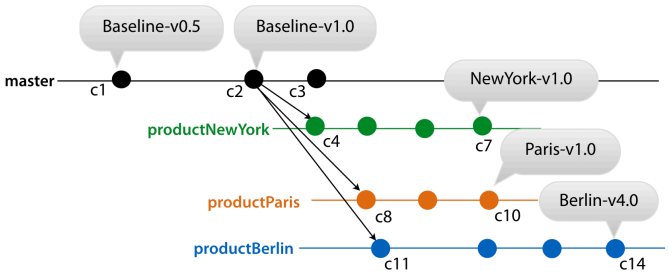

Figure 3.2: *WeatherStationSPL* branching model: the *master* branch holds the core-assets baselines from where SPL products are branched off.

questions is addressed through a questionnaire delivered to SPL practitioners (Section 3.5),

- RQ2. We develop *CustomDIFF*, a DW approach that uses *Git* as the operational system from where fact data is obtained, and *pure::variants* as the SPL framework (Section 3.6),
- RQ3. We resort to Alluvial diagrams to visualize the customization effort at a glance. These diagrams are a type of flow diagrams. Here, the flow stands for the customization effort that goes from core-assets to SPL products where customization was needed (Section 3.7).

Next Section illustrates the challenge with a motivating scenario.

3.3 Motivating scenario

As an example, consider the *WeatherStationSPL*, an SPL for building web-based applications for weather stations. We borrow this example from the experimental material provided by *pure::variants*, a well-known industrial tool for developing SPLs [pur]. Figure 4.2 depicts a certain stage in the *WeatherStationSPL* journey. So far, this SPL has undergone two core-asset baseline release at the *master* branch, i.e ,*Baseline-v0.5* and *Baseline-v1.0*. The latter holds seven features realized through 30 code assets (see Table 3.1). Figure 3.3 shows a snippet of the core-asset *sensors.js*. This snippet shows two variation points, i.e. VP-1 and VP-2. In *pure::variants*, a variation point starts with the opening directive *//PV:IFCOND,* and ends with a closing directive *//PV*:*ENDCOND* [2]. Hence, VP-1 body comprises lines 24 to 49, whereas VP-2 expands along lines 30 to 46.

From *Baseline-1.0*, three products are branched off: *productParis*, *productBerlin* and *productNewYork*. Let us consider that some urgent needs arise that prevent customers from waiting to the next SPL release. This moves us to the *grow-and-prune* process:

[2]These variation point patterns only hold for code files. For example, in XML and HTML files, variable elements are annotated in an attribute called *condition*.

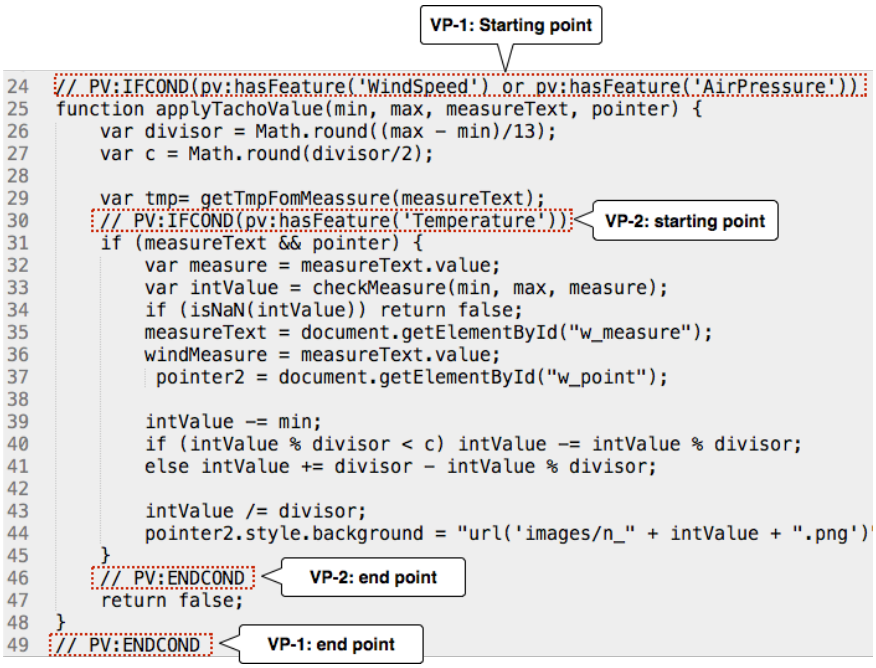

Figure 3.3: *Sensors.js* core-asset at Baseline-v1.0. The snippet shows two variations points. VP1 applies when either *WindSpeed* or *AirPressure* are selected. VP2 applies for *Temperature*. Notice how VP2 is scoped within VP1.

- **Grow**. Product specifics can be promptly considered in the products' branches by adjusting core-assets to product specifics, delivering the new product version on time (e.g. *Paris-v1.0*),
- **Prune**. During SPL consolidation, domain engineers prepare for the next SPL release. Analyzing how core-assets have been used by products becomes a main stepping stone in ascertaining reuse opportunities, and deciding which feature upgrades are going to be supported in the next SPL release.

The *grow-and-prune* is an evolution model, for moving SPL approaches from a state S0 to a better state S1. This "better state" is to be (partially) obtained out of product customizations that, independently of one another, have already moved to a "better state" making the core-assets C0 evolve into customized assets C1. Hence, pruning requires to analyse how core-assets are being customized, i.e. looking at the difference between core-assets (kept in the *master* branch) and namesake assets once customized by products (kept in the product branches). This is commonly performed one file at a time: *diff(C0.file, C1.file)*. Back to the example, Figure 3.4 illustrates the case for *sensors.js* using the DIFF utility in the unified format [uni]. For each change hunk, the outcome indicates: the hunk header (i.e. starting and ending line numbers together

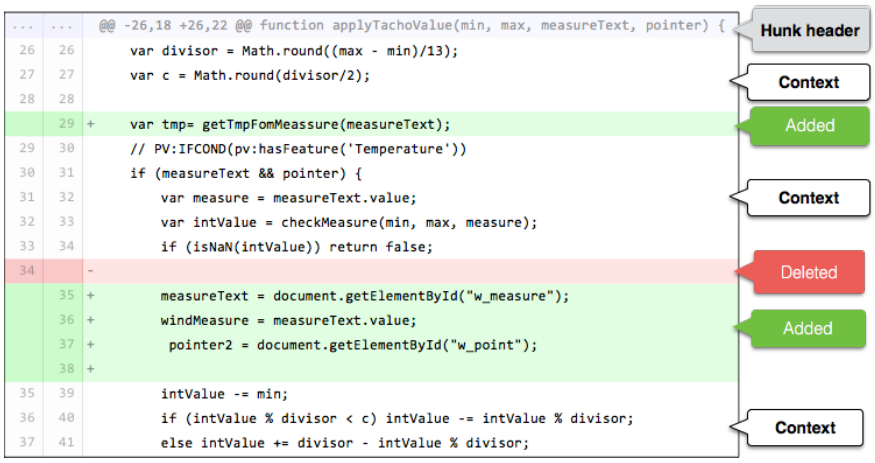

Figure 3.4: Traditional DIFF visualization: differences of file *sensors.js* between the one in the Master branch (core-assets) and the one in the *productBerlin* branch.

Parent Feature	Feature	Description
Sensors	AirPressure	The weather station system measures the air pressure and displays in a pressure gauge
	Temperature	The weather station system measures the air pressure and displays in a thermometer gauge
	WindSpeed	The weather station system measures the wind speed and displays it in a speed gauge
Warnings	Gale	The weather station alerts the user when the wind speed value surpasses the maximum.
	Heat	The weather station alerts the user when the temperature value surpasses the maximum.
Languages	English	The weather station front-end texts are available in English
	German	The weather station front-end texts are available in German

Table 3.1: *WeatherStationSPL* features at Baseline-v1.0.

with the *heading* of the function the change hunk is part of), the context (i.e. the three nearest unchanged lines that precede and follow the change), the added lines (denoted by a plus sign with a greenish background) and the deleted lines (denoted by a minus sign with a redish background).

However, *sensors.js* is just one of the thirty files this SPL encompasses. And this thirty files might potentially suffer changes by any of the three products. This implies that domain engineers will potentially need to scan 30 x 3 DIFFs. Now move to the Danfoss Drives SPL, which holds over 10,000 core-assets and 20 products. And this is just to get the raw data, i.e. the lines of code (LOC) that have been changed. These LOCs need next to be cleaned-up and aggregated in meaningful analysis terms (i.e. features and products). In short, manually conducting DIFFs does not scale up. Fortunately, mechanisms already exist that might help: data warehouses.

3.4 A Data Warehouse approach to customization analysis

Data warehouse (DW) is a collection of decision support technologies, aimed at enabling knowledge workers to make better and faster decisions [KR02]. Implementation wise, DWs are systems used for reporting and data analysis. A main component is a central repository of integrated data from one or more disparate sources (a.k.a the **operational systems**). They store current and historical data in one single place that are used for creating analytical reports for decision making. To this end, raw data is conducted through an **Extract, Transform, Load (ETL)** process that ends up in the data being arranged along facts (i.e. the aspects to be measured) and dimensions (i.e. the ways measures are going to be broken down). The combination of facts and dimensions is called a **Star Schema** that results from **dimensional modeling**.

Dimensional Modeling supports analysis of a process by modeling how it is measured [KR02]. Here, the process to be measured is

> *the customization involved in tuning reusable artifacts for product specifics*.

This process it is to be measured through the number of lines of code (LOC) being added/deleted. That is, LOCs are regarded as manifestations of the customization effort. This data is to be obtained through the SPL's Version Control System (VCS) (e.g. *Git*), our operational system. Specifically, during the ETL process, a DIFF is worked out between the namesake artifacts of the core-asset and the product at hand. Differences stand for the adjustments (i.e. LOCs) introduced to account for product specifics.

However, LOCs are too fine-grained, and fail to provide a holistic view of the customization effort at a glance. This is when Dimensional Modeling comes into play. File-based LOC changes are the finest grain of data but it can be rolled up to various levels of dimensionality till reaching coarser grains, such as, "feature" or "product". In short, DW is the means to move from traditional *diff(aFile, aFile)* to *diff(aFeature, aProduct)*.

We fleshed out this vision through *CustomDIFF*, an ETL and visualization tool for SPL customization analysis for *Git* as the operational system, and *pure::variants* as the SPL framework. A video showing *CustomDIFF* at work is available at `https://tinyurl.com/ycjhwzpc`. Next sections delve into three key notions in DW projects: Requirement Analysis, Dimensional Modeling and Reporting Tools.

3.5 Requirement analysis

As in all software projects, a critical phase in the DW lifecycle is the Requirement Analysis phase. The predominant objective of this phase is to identify organization goals and elaborate requirements that could measure organization performance [AYD13]. Here, we resort to a goal-oriented approach described in Mazon et al. [MPT07] where data needs are obtained out of organizational goals. Specifically,

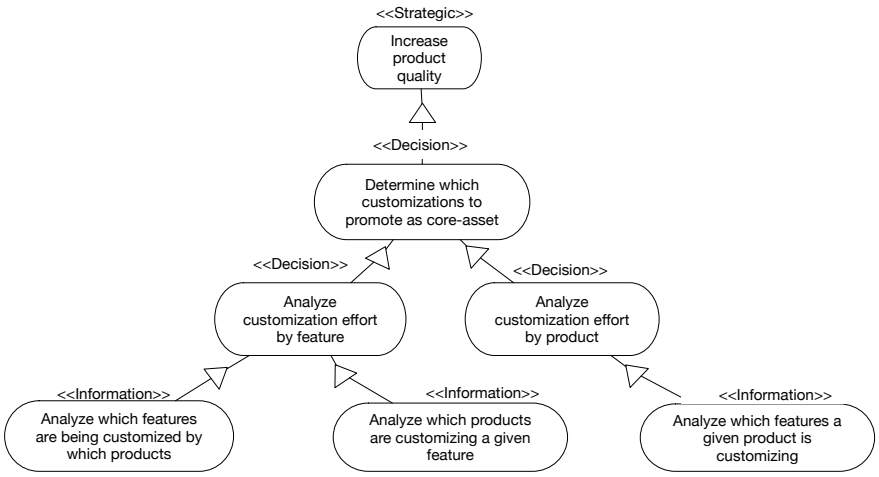

Figure 3.5: Goal, decisions and information needs for customization analysis. Notation along the profile introduced in Mazon et al. [MPT07] for DW requirements.

goals are classified as strategic, decisional and informational. Next, we apply this approach for the case of customization analysis (see Figure 3.5). Other aspects of the SPL evolution process are outside the scope of this work.

Strategic goals. Strategic goals are thought as "changes from a current situation into a better one" [MPT07] (e.g. "increase sales", "increase number of customers"). Their fulfillment causes an immediate benefit for the organization. For SPLs, strategic goals can be reducing time-to-market, increasing product quality, etc.

Decision goals. Strategic goals are detailed out into decision goals that are more actionable (e.g. "open new stores"). For SPLs, strategic goals require of an "updated core-asset base" that is periodically upgraded. In this setting, we can make the decision to "determine which customizations to promote as core-assets". This decision might be refined by differentiating between two focuses:

- feature focus, when deciding when and what customizations are to be included in the next SPL release. In Danfoss, a specific body exists (known as the Change Control Board (CCB)) who decides the pace at which the SPL evolves, i.e. which feature upgrades are to go into the next SPL release.

- product focus, to assess whether the importance of the customer or the revenue coming from a product, might favor prioritizing the customizations coming from a given product. In Danfoss, each product-development project has its own committee that determines whether a request for development will go to the CCB [DSB05].

The CCB synchronizes the requests from different projects and decides whether and which of the requests will be honored. To this end, the CCB should balance the upgrade

costs (estimated development cost and developer agenda availability) vs. the upgrade benefits (how many and which products will benefit from the upgrade). In addition, upgrades might have different degrees of urgency from as-soon-as-possible (e.g. bug fixes) to desirable (e.g. new fancy functionality). Finally, and depending on the SPL size, feature management might be split among distinct domain engineers. Each feature (or set of related features) is up to a team which is in charge of coding, debugging and upgrading the features at hand [DSB05]. Here, SPL engineers might need to track what, how and where have products changed "their" features. This moves us to the information goals.

Information goals. These goals aim to capture which specific information could help to obtain the strategic goals. As for the customization effort, two main variables are involved: features and products. This permits to tackle the analysis through a three-fold perspective: (1) holistic (e.g. which features are being customized by which products); (2) feature-focused (e.g. for a given feature, which products are customizing it); and (3), product-focused (e.g. for a given product, which features were necessary to customize). In addition, the customization effort admits different levels of granularity: the artefacts being affected by the customization effort (#); the number of lines of code (LOC) involved in the customization; or the code itself that supports the customization.

These two dimensions (i.e. perspective and grain) permit to systematize the kind of questions analysts might face (see Table 3.2). To validate the importance of these questions for our Decision goal, we conducted a survey among practitioners. Participants were selected who had at least one-year experience on SPLs. Eight practitioners turned up where three have 10 years of experience while the other five accounted for 9, 7, 6, 3, and 1 year of experience each. Though practitioners all come from the same company, they might have different duties, and hence, they were requested to provide their opinion from their specific "SPL plot". For instance, two domain engineers (application engineers) responsible for distinct features (products) might give different answers depending on how their features (products) behave regarding customization needs.

The questionnaire was first checked with two researchers for clarification and understanding purposes. Next, practitioners were requested to indicate the extent to which the agree with each of the statements along a LIKERT scale from 1 ("Strongly disagree") to 5 ("Strongly agree"). Table 3.2 shows the results. Some comments are due:

- perspective wise, both feature-focused and product-focused are similarly rated. The highest rated questions are "for the feature F1, which products are customizing it" (avg. 4) and its sibling, i.e. "for the product P1, which are the features being customized" (avg. 4.12)

- grain wise, quite an unexpected result: intermediary grains are ranked down. Analysis needs seem to be biased towards gaining either a general overview of the customization effort (i.e. *[#]* questions) or instead, diving into the nitty-gritty code details (i.e. + *code* questions)

So far, these information needs are fulfilled by directly peering at the code. We requested the participants to also indicate how much time they currently need to solve

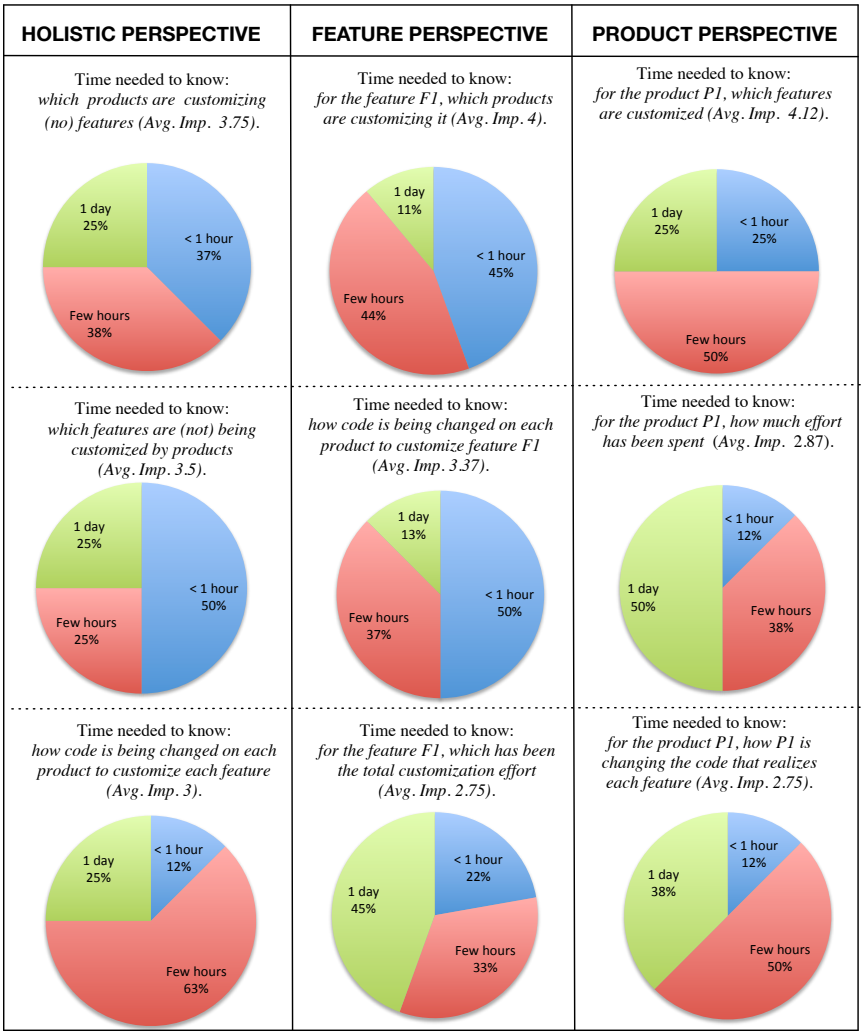

Figure 3.6: Time spent on solving information needs for Customization Analysis. The question description is followed by the average importance obtained from the questionnaire in Table 3.2.

Holistic perspective	Item: "I consider important to know ...	Likert scale					Avg.
		1	2	3	4	5	
(Fi, Pi) [#]	... which features are (not) being customized by products	0	1	3	3	1	3.5
(*, Pi) [#]	... which products are customizing (no) features	0	1	2	3	2	3.75
(Fi, *) [LOC]	... how much effort (i.e. LOC) has been spent on customizing each feature, in total, no matter the product	2	1	4	0	1	2.87
(Fi, Pi) [LOC]	... how much effort (i.e. LOC) *each* product is spending on customizing *each* feature	2	2	2	1	1	2.62
(Fi, Pi) + code	... how code is being changed on each product to customize each feature	2	0	2	4	0	3
Feature perspective	**Item: "I consider important to know ...**	**Likert scale**					**Avg.**
		1	**2**	**3**	**4**	**5**	
(F1, Pi) [#]	... for the feature F1, which products are customizing it	0	0	1	3	3	4
(F1, *) [LOC]	... for the feature F1, which has been the total customization effort	2	1	3	1	1	2.75
(F1, Pi) [LOC]	... for the feature F1, how much effort (i.e. LOCs) each product is spending on customizing it	2	1	3	1	1	2.75
(F1,Pi) + code	... how code is being changed on each product to customize feature F1	1	2	1	1	3	3.37
Product perspective	**Item: "I consider important to know ...**	**Likert scale**					**Avg.**
		1	**2**	**3**	**4**	**5**	
(Fi, P1) [#]	... for the product P1, which features are customized	0	1	1	2	4	4.12
(*, P1) [LOC]	... for the product P1, how much effort has been spent on customization, no matter the feature	2	1	2	2	1	2.87
(Fi, P1) [LOC]	... for the product P1, how much effort (i.e. LOC) has been spent on customizing each feature	2	1	3	1	1	2.75
(Fi, P1) +code	... for the product P1, how P1 is changing the code that realizes *each* feature	2	1	3	1	1	2.75

Table 3.2: Rating the importance of information needs along a 5 point LIKERT scale.

these information needs. The outcome is depicted in Figure 3.6 for the highest rated queries. Ultimately, these figures vindicate the effort of providing dedicated tools to customization analysis.

3.6 Dimensional modeling

The previous section uncovers information needs for customization analysis. This provides the grounds for coming up with the DW Star Schema, a blueprint of the database schema that will eventually support the customization analysis. Figure 3.6 depicts the Star Schema for *CustomDIFF*.

The Fact table. A main decision is the *grain* at which the customization effort is to be captured. In Dimensional Modeling, the grain is the fundamental atomic level of

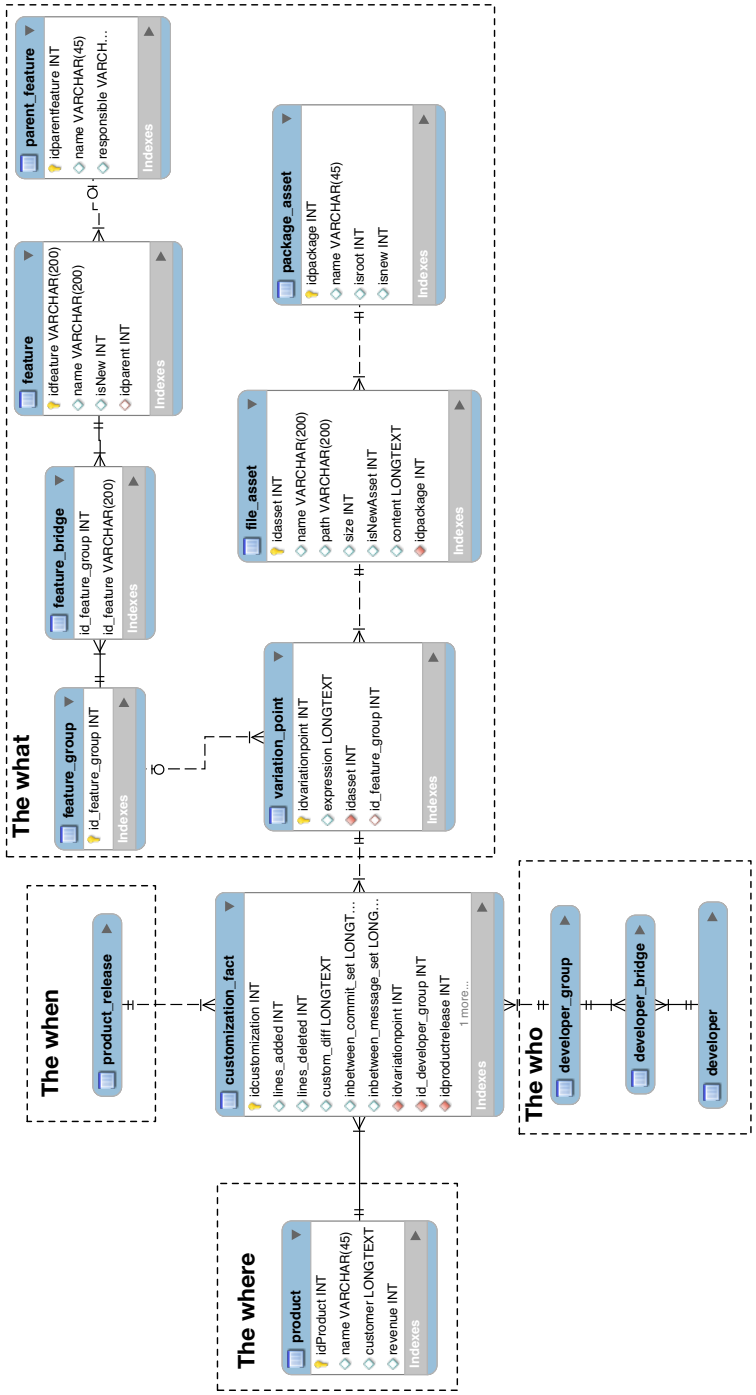

Figure 3.7: Start/Snowflake schema for *CustomDIFF*.

data to be represented in the Fact table. For our purposes, we consider a fact to be

> *the consecutive deletion/addition of LOC for files pertaining to a given release ("when") conducted by developers ("how") in order to customize the body of a given variation point ("what") to account for the specifics of a given product ("where")*

These facts are obtained by working out a DIFF between the namesake artifacts of the core-asset and the product at hand. For the DIFF depicted in Figure 3.4, two facts would be obtained. Fact #1 would stand for the change introduced in line 29, whereas Fact #2 would correspond to those changes introduced in lines 34-38. Fact properties include: the number of lines added, the number of lines deleted or the actual code being changed (*custom_diff*).

Facts are the finest grain of the customization effort. Obtaining a higher vision of the customization effort requires these facts to be aggregated along different dimensions: "the what" (i.e. the VP being affected by the customization), "the where" (i.e. the product in which the customization took place), "the when" (i.e. the time of the product release), and "the who" (i.e. developers who conducted the customization).

The *"what"* dimension (*variation_point* table). The customization effort (i.e. addition/deletion of code lines) takes place within a context: the body of the variation point (VP). Products derived from the same core-assets will be able to touch the same VPs. VPs are embedded within *file_asset*s. Finally, file assets might be arranged along packages (basically, folders). This conforms *the asset hierarchy*. That is, tables "customization_fact", "variation_point", "file_asset" and "package_asset" (see Figure 3.7) all hold a one-to-many relationship that permits to gradually aggregate customization measures along coarser-grained assets.

Besides the asset hierarchy, another aggregation criterium are features. Variation points include a boolean *expression* that checks out feature presence. A customization effort might then impact a *feature_group*[3]. Along good practices on dimensional design, this is captured using a "bridge table"[4] (*feature_bridge*). This permits analysis to be conducted at the level of single features. However, Danfoss practitioners observed that for large number of features, it would be convenient to undertake the analysis at the parent-feature level. Parent features gather together a set of related features, e.g. *'AirPressure', 'Temperature'* and *'WindSpeed'* belong to the same parent feature *'Sensors'* (see Table *3.1*). Hence, we also include information about the *parent_feature*. Summing it up, customization efforts (i.e. facts) are scoped by VP expressions which might refer to different features, which, in turn, might be clustered along parent features. This conforms *the feature hierarchy.*

[3]Worth noticing, VPs might be nested. Figure 3.3 illustrates this situation for VP-1. The frequency of these situations (feature tangling and feature nesting) is being studied in [HZS+16, ZBP+13]. This begs the question of what are the features affected by a customization effort inside VP-2's body. For our purposes, we include all features: those appearing directly in the VP body (i.e. *'Temperature'*) as well as those "inherited" from containing VPs (i.e. *'WindSpeed'* and *'AirPressure'*).

[4]An alternative design is to flatten the multi-valued attribute by including a column for each of the different values: a boolean column will be included for each possible feature, setting it a true if the feature at hand participates in the VP expression. This however will tight the dimensional design to the current feature model. Adding (or deleting) features would need to be propagated to the *variation_point* table. Since we are considering SPLs in an early stage of development where changes in the feature model are likely, we rather stick to the "bridge table" option.

The *"where"* dimension (*product* table). Customization efforts are contextualized within products. Product characterization (e.g. customer, contact details, priority, etc) are not addressed in this work.

The *"when"* dimension (*product_release* table). Customizations are committed at a given time. However, we do not consider in-between releases but just final releases where the product is ready to be delivered. This increases the confidence that the customization effort being measured, has been appropriately tested before being disclosed to customers. Therefore, the customization effort is for changes already available at the product release. It can be argued that this does not account for all the effort that goes till reaching this final state. That's true. But this would measure more a kind of productivity effort rather than the amount of change. Nevertheless, property *inbetween_commit* for the fact table, collects the ID for those intermediary releases, just in case.

The *"who"* dimension (*developer_group* table). This dimension collects data about the engineers that conduct the customization. Since more than one person might be involved, we capture the notion of group that is next broken down in each of its members (*developer*). We do not further tackle this dimension here.

Figure 3.6 depicts the main tables that result from Dimensional Modeling. These tables are populated out the SPL's Git repository through the ETL process. This process is responsible for pulling data out of the operational systems and placing it into a data warehouse. Most DWs combine data from different source systems. For the time being, however, we stick to a single data source: the SPL's Git repository. Appendix A provides details of *CustomDIFF's* ETL process.

Once the data is in placed, Reporting Tools help to interactively explore the customization-effort dimensions. Broadly speaking, Reporting Tools can be regarded as a continuation of existing DIFF utilities. While these utilities currently permit to assess the size of the change at the level of files, DW Reporting Tools would permit this assessment to take place at the level of features and products that might potentially encompass tens, if not hundreds, of files. Next section introduces such a reporting capability for *CustomDIFF* with a focus on the "where" dimension.

3.7 Reporting tools

As captured in the Requirement Analysis stage, Decision goals might be "product-focus" ("the where") or "feature-focus" ("the what"). We believe Alluvial diagrams might help to convey this double focus. Alluvial diagrams are a type of flow diagram originally developed to represent how multiple groups relate to one another across several variables. We resort to this visualization to convey the relationship for the two main dimensions of the customization effort: "the where" and "the what".

Figure 4.7 shows the case for our running example. Each dimension is assigned to a vertical axe: the feature axe (right) vs. the product axe (left). Values are represented with blocks on each axis. The height of a block represents the size of the customization effort for this feature/product, and the height of a stream represents the degree of the customization effort contained in both blocks connected by the stream field. Looking at Figure 4.7, we can promptly appreciate how the parent feature

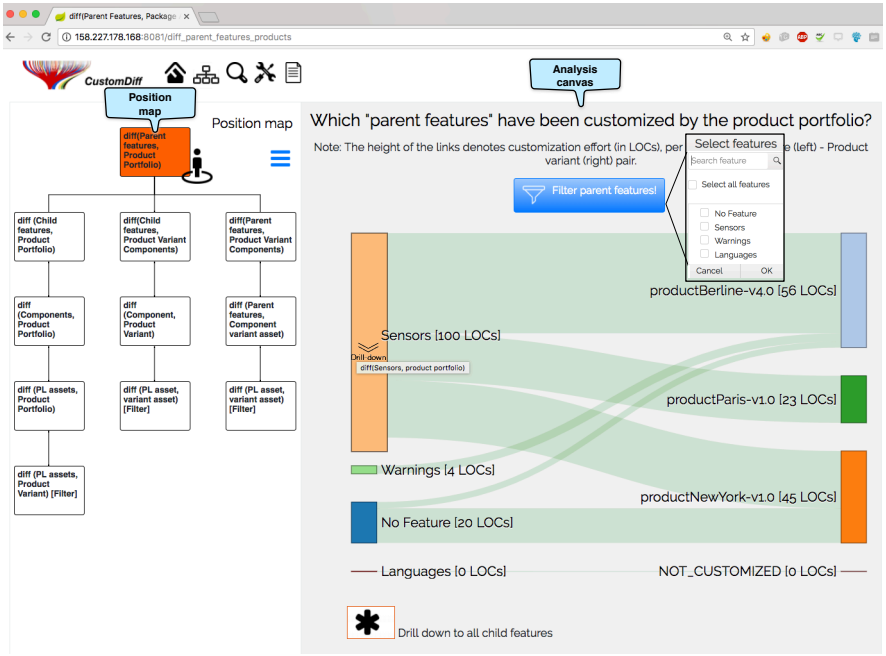

Figure 3.8: *CustomDIFF* screenshot: Position map (left) and Analysis canvas (right). The Analysis canvas displays the alluvial diagram to assess the customization effort for parent-features (left axe) and products (right axe). Customizations conducted outside VP bodies (impacting no feature) are collected under the name *"No Feature"*.

Sensors is being customized by *productBerlin*, *productParis*, and *productNewYork*, with product *productBerlin* being the one with the largest customization effort. Alluvial diagrams also help to promptly appreciate which variables are more clustered (fewer, wider streams) and which are more distributed (more, narrower streams). For instance, *Sensors* is being evenly customized in the three products, whereas *No Feature* is being mainly associated with *productNewYork*. Next, we provide different analysis scenarios. Readers are encouraged to access the running example on-line at `http://158.227.178.168:8081/analysis`.

Figure 4.7 shows the extent of the customization effort at the level of parent features and products. But finer-grained details might be needed. Back to the Star Schema in Figure 3.7, we notice "the where" dimension accounts for two hierarchies, namely, the asset hierarchy and the feature hierarchy. Developers can move up and down each of these hierarchies by clicking on the respective blocks. An example follows for the feature axe:

- when in Figure 4.7, clicking on the *Sensors* block, drills down into its child features. The outcome is depicted in Figure 3.9 (top) where *Sensors'* child features become blocks on the left axe,

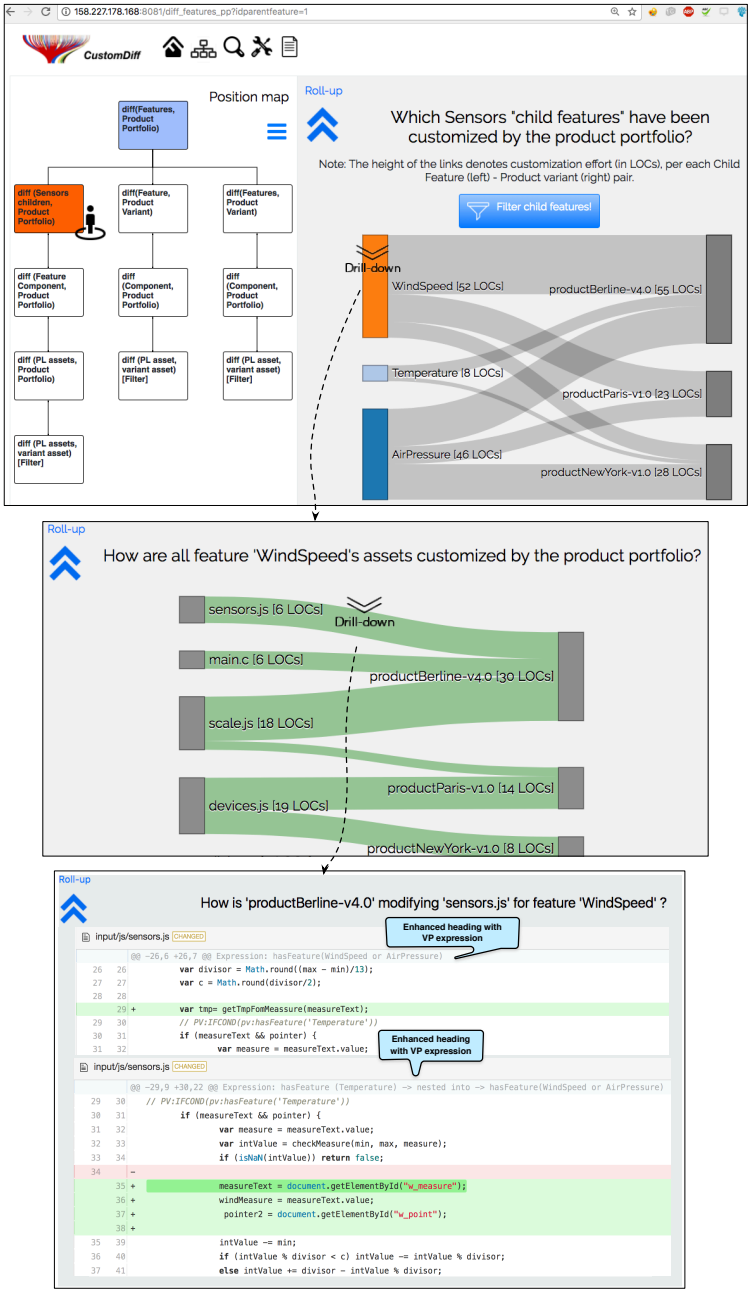

Figure 3.9: Drilling-down scenario. Breaking down customization efforts for *Sensors* by *Sensors'* child features (top); next *WindSpeed's* assets (middle), and finally raw facts (bottom).

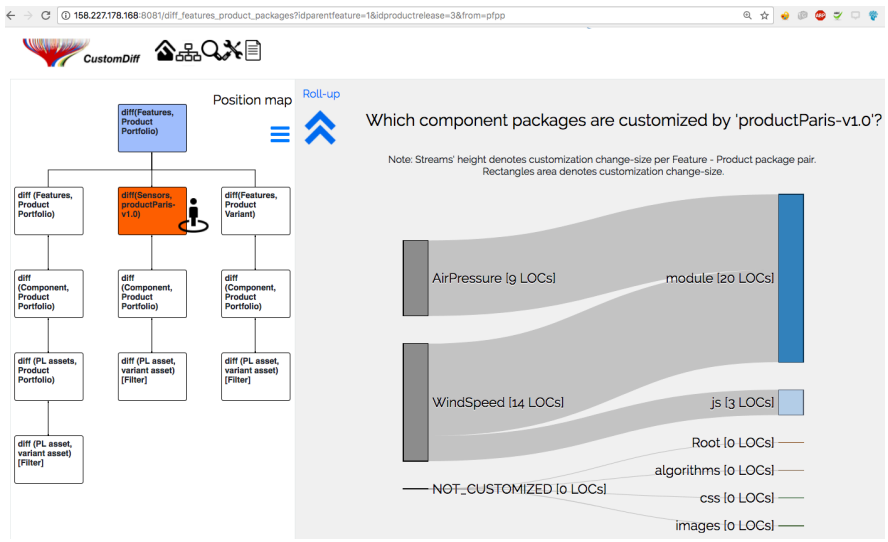

Figure 3.10: Stream-based drill down. Simultaneously breaking down the customization effort for *Sensors* and *productParis'* packages.

- when in Figure 3.9 (top), clicking on a feature block, e.g. *WindSpeed*, drills down into the artefacts impacted by this feature[5]. Figure4.7 stuffed(middle) depicts the output where the customization effort is broken down by *WindSpeed*'s files.
- when in Figure 3.9(middle), clicking on a stream, e.g. *sensors.js&productBerlin-v4.0*, drills down into the code level diff. Now, the customization effort is broken down by the lines of code that *productBerlin* changed in *sensors.js*. This information is displayed mimicking traditional DIFF outputs but where VPs are shown as separated hunks (see Figure4.7 (bottom)).

The latter example illustrates how *streams* also denote customization efforts, but this time involving a value for each of the axes, e.g. *(Sensors, productBerlin)*. By clicking on the stream, the associated customization effort is broken down simultaneously for the two variables affected. Figure 3.10 shows the case for *Sensors* and *productParis,* which are decomposed into features and packages, respectively. Next, we introduce some improvements that resulted from previous formative evaluations of *CustomDIFF*.

Enhancing DIFF context description DIFF traditional visualization includes the so-called context, i.e. the three nearest unchanged lines that precede and follow the change (see Figure 3.4). The context serves as a reference to locate the changed lines' place in a modified file. However, this might not be enough for SPLs. In SPLs, code is stuffed with variation points (VPs) that determine when the associated body is to be

[5]Note that there is one level behind, i.e. component package level. For the sake of understanding, we omit this step.

included. Hence, VPs are a main contextual information for changes. Unfortunately, VPs are realized as comments, which can be placed far away from where the change has occurred, and hence, might not show up in the DIFF context, depriving engineers from this information. This is the case for the change introduced in line 29 in Figure 3.4. Lines 26, 27, 28, i.e. the context, do not include the VP annotation.

CustomDIFF adds VP expressions as part of the DIFF visualization. Figure 3.9 (bottom) shows the same case that Figure 3.4, but now information about the enclosing VP is included into the headings of each hunk. Note how the DIFF is now split into two hunks, one for each customization fact. One hunks corresponds to the change in line 29 under the scope of VP-1 *(WindSpeed or AirPressure)*. The other hunk stands for the changes in lines 34-38 under the scope of VP-2. Notice that VP-2 is nested within VP-1. This is reflected in the hunk's heading along the pattern*: <enclosing VP> "–> nested into –>" <enclosed VP>*. Notice that all this information is pre-computed in the fact table, specifically the hunk is stored into the *custom_diff* property.

The position map Engineers might get disoriented when moving up and down the hierarchies. To ease location (where am I?), a left hand-side collapsable panel is deployed besides the Analysis canvas. The map pinpoints the current grain in the dimension hierarchy (see node with orange background in Figure 4.7) as well as the grains already visited (see node with blue background in 3.9 (top)).

Feature-based Filtering Engineers might not be interested in the whole set of (parent) features but just a subset. The feature-based filter (see Figure 4.7) permits engineers to select those features they are interested in using a feature-diagram similar to the one displayed in *pure::variants*. The Analysis canvas will depict the alluvial diagram just for the selected set of features.

3.8 Evaluation

By moving from the traditional DIFF to *CustomDIFF*, we aim at making customization analysis more efficient and effective. Efficiency wise, DWs outperform DIFF, insofar as results are precomputed and access is conducted through performant SQL engines. But what it rests to be seen is whether DW in general, and *CustomDIFF* in particular, is effective, i.e. they help to satisfy the information needs of SPL analysts. To this end, this section attempts to predict the acceptability of tools, such as, *CustomDIFF* by applying the Technology Acceptance Model (TAM) [Dav89].

TAM proposes that the readiness of a user to use (or not to use) a new technology is determined by her attitude towards the technology. This attitude is influenced by two beliefs which are *perceived usefulness* and *perceived ease of use*. Perceived usefulness is defined as "the degree to which a person believes that using a particular technology would enhance his or her job performance" [Dav89]. On the other hand, ease of use refers to "the degree to which a person believes that using a particular system would be free of effort" [Dav89]. According to the theory of reasoned action [FA75], these constructs are strongly correlated to the intention of actually using the technological innovation. No matter how easy to use a tool can be; if the tool is not perceived as

	Sensors			Language		Warning		No
	Air Pressure	Wind Speed	Temperature	German	English	Gale	Heat	feature
Product Berlin	19 LOCs	30 LOCs	0 LOCs	0 LOCs	0 LOCs	2 LOCs	2 LOCs	3 LOCs
Product Paris	9 LOCs	14 LOCs	0 LOCs	0 LOCs	0 LOCs	0 LOCs	0 LOCs	0 LOCs
Product NewYork	18 LOCs	8 LOCs	2 LOCs	0 LOCs	0 LOCs	0 LOCs	0 LOCs	17 LOCs

Table 3.3: Experiment: products and customization effort per feature.

useful by its users it would not be used. The opposite also holds. A technology might be very useful; but if the tool is cumbersome and hard to use, users would not use it either. Hence, we decided to use both constructs. Therefore, we aim at

> analyzing the use of *CustomDIFF* for the purpose of evaluating its *usefulness* and *ease of use* with respect to conducting c*ustomization analysis* from the point of view of SPL practitioners in the context of annotation-based SPLs.

3.8.1 Participants

Customization analysis is to be entrusted not to newcomers, but to those with a reasonable exposure to SPL engineering. In addition, customization analysis allows for different perspectives (feature vs. product) that might depend on the subjects' role, balanced between domain engineering and product engineering. On these grounds, participants were selected among Danfoss developers with at least one year experience and having heterogenous roles: 1 product-release manager, 3 software developers that accomplish both domain and application engineering task, and 2 more that also act as code reviewers before changes are integrated into the integration branch. Participants' average expertise on the SPL was 7 years.

3.8.2 Training examples

Before delivering the questionnaire, it is most important for participants to be exposed to the system under study (i.e. *CustomDIFF*). The faithfulness of these sample cases w.r.t the real practice, is paramount for participants to correctly validate the tool, and for researchers to assess the validity of the experiment. This subsection describes these sample cases.

Sample cases were selected among those ranked highest during the Requirement Analysis stage (see Table 3.2). The *WeatherStation* SPL was used as the running example (see Section 3.3). This SPL is included in *pure::variants* [pur] experimental material, and hence, participants were already familiarized with it. This permits participants to focus on customization issues rather than on mixing up with the SPL domain itself.

For the evaluation, three product variants were created: *productParis*, *productBerlin* and *productNewYork,* each with a set of customizations (see Table 3.3). For instance, *productBerlin* customizes *AirPressure, WindSpeed, Gale* and *Heat* by changing 19, 30, 2, and 2 LOCs, respectively. Both, core-assets and products, reside in a GIT repository. *CustomDIFF* taps into this repository. We deployed *CustomDIFF* online at `http://158.227.178.168:8081/analysis` (open for inspection).

Upon this setting, participants were requested to conduct distinct tasks along the different perspectives identified in Section 3.5. For each perspective, two types of tasks were performed:

- pinpointing a given artefact (either a feature or a product)
- conducting roll-up and drill-down along "the where" dimension, either the feature hierarchy or the asset hierarchy

Next, we list the tasks:

- **Feature Perspective: Analyze the evolution of feature AirPressure**
  - *Task 1.1: Which products are customizing the AirPressure child-feature?*
  - *Task 1.2: Analyze how the code that realizes AirPressure has been changed by the product portfolio.*
- **Holistic perspective: Analyze how the whole set of features is being customized**
  - *Task 2.1. Which parent features are not customized by the products?*
  - *Task 2.2. Analyze how productParis is changing the implementation of the Sensors parent feature*
- **Product perspective: Analyze how product Berlin has evolved from the core-assets**
  - *Task 3.1. Which parent-features is productBerlin customizing?*
  - *Task 3.2. Analyze how the implementation of productBerlin has evolved.*

3.8.3 Procedure

Session 1: Introduction to *CustomDIFF* (1h 45'). *CustomDIFF's* rationales and operations were introduced to the participants with the help of the *WeatherStation* SPL. Next, *CustomDIFF* was used for the Danfoss Driver SPL. Participants made questions during and after the presentation.

Session 2: Hands-on *CustomDIFF* (1h 30'). *CustomDIFF* was evaluated w.r.t. *usefulness* and *ease of use*. First a running example was conducted (see previous subsection) where participants were exposed to different information needs. Next, the evaluation questionnaire was deployed on-line for the participants to assess *CustomDIFF's usefulness* and *ease of use.* Due to agenda constraints, participants were divided into two group, with 2 and 4 participants each. During the sessions, a researcher was observing participants' interactions with the tool. Participants raised questions, doubts and comments that were noted down by the researcher.

Perspect.	Item: CustomDIFF was useful to determine ...	P1	P2	P3	P4	P5	P6	Avg.	St. Dev.
Feature	...which products are customizing the child-feature AirPressure?	6	4	6	6	6	6	5.67	0.82
	...how is each product customizing the code that realizes the AirPressure feature	6	4	6	6	7	4	5.5	1.22
Holistic	...which parent-features are not customized by the products	7	5	6	6	6	6	6	0.63
	...how productParis is changing the implementation of Sensors parent-feature	6	5	6	6	7	5	5.83	0.75
Product	...which parent-features is productBerlin customizing	7	4	6	6	6	6	5.83	0.98
	...how the implementation of productBerlin has evolved	6	4	6	6	6	4	5.33	1.03
Total		6.33	4.33	6	6	6.33	5.17	5,.9	5.43

Table 3.4: *CustomDIFF's* perceived usefulness.

3.8.4 Results

Usefulness. Usefulness was evaluated w.r.t. information findability, i.e. to what extent does it help users find the required information needed for customization analysis. Table 3.4 gathers the results of a questionnaire where agreement with statements is rated along a LIKERT scale that ranges from 1 ("Strongly disagree") to 7 ("Strongly agree").

Participants rated *CustomDIFF* with a total average of 5.69. As for the type of questions, i.e. artefact pinpointing vs. hierarchy drill down, i.e. the former is consistently punctuated slightly higher.

Ease of use. Davis' template was used for evaluating this aspect (see Table 3.6). Participants rated *CustomDIFF* with an avg. 5.44. This general template was extended to assess *CustomDIFF* specific mechanisms (see Table 3.6): the use of alluvial diagrams (5.84), the parent-feature dimension (5.5), the component-based dimension (5.33), the VP-enriched context *DIFF* (5.5), the feature-based filtering utility (5.83) and the analysis position map.

3.8.5 Discussion

Preliminary evaluation shows promising results for *CustomDIFF*. The aspects most highly rated include:

- the easiness to find the *DIFF* between products and features,
- the overall picture provided by alluvial diagrams.

Item:	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**Avg.**	**St. Dev.**
Learning to operate *CustomDIFF* would be easy for me	6	4	7	6	6	6	5.83	0.98
I would find it easy to get *CustomDIFF* to do what I want it to do	5	4	6	6	5	4	5	0.89
My interaction with *CustomDIFF* would be clear and understandable	6	4	6	6	6	5	5.5	0.84
I would find *CustomDIFF* to be flexible to interact with	5	4	6	5	6	5	5.17	0.75
It would be easy for me to become skillful at using *CustomDIFF*	7	4	6	6	6	5	5.67	1.03
I would find *CustomDIFF* easy to use	6	4	6	6	6	5	5.5	0.87
Total	5.83	4	6.16	5.83	5.83	5	5.44	0.89

Table 3.5: *CustomDIFF's* perceived ease of use. Evaluation along a LIKERT scale from 1 (total disagreement) to 7 (total agreement).

Item: I would find ...	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**Avg.**	**St. Dev.**
...alluvial diagrams useful for grasping customization effort	6	5	6	6	6	6	5.84	0.41
...the parent-feature dimension useful to abstract away from individual features	6	5	5	5	6	6	5.5	0.58
...the component-based dimension to abstract away from files	5	5	6	5	6	5	5.33	0.52
... the VP-enriched context Diff to easy locate change placement into code	6	5	7	6	5	4	5.5	1.05
... feature-based filtering utility useful to easy focus	7	4	6	6	6	6	5.83	0.98
...the position map useful to position myself during customization analysis	6	4	5	6	5	6	5.33	0.82
Total	6	4.66	5.83	5.67	5.67	5.5	5.55	0.72

Table 3.6: *CustomDIFF's* specific utilities. Evaluation along a LIKERT scale from 1 (total disagreement) to 7 (total agreement).

Subsequent discussions with participants also helped to identify two additional use cases where *CustomDIFF* can help:

1. as a code-review tool. When product customizations are to be promoted (i.e. integrated into the SPL core-assets), code reviewers need to ensure that these changes do not affect other products in the first place. This is ensured by surrounding the customization with a variation point that includes a brand new child feature that is initially enabled only for the product from where the customization was originated. *CustomDIFF* might help to trace if this practice is followed.

2. as an impact analysis tool for integrating *frozen* products into the SPL. *Frozen* products are those derived from the SPL core-asset base several years ago but which evolved independently afterwards. At a certain point, the decision is made that a *frozen* product needs to be upgraded to the current SPL release. This decision needs to weight the effort needed. This can be achieved using *CustomDIFF*.

3.8.6 Threats to validity

Construct Validity refers to the degree of accuracy to which the variables defined in a study measure the constructs of interest. Here, the constructs are *usefulness* and *ease of use*, while variables are the items of the used questionnaires, and the answers participants provide to the tasks. Usefulness was assessed in terms of the fulfillment of information needs. These information needs were carefully selected to be paradigmatic of the different scenarios that might rise during customization analysis, specifically, those scenarios that were ranked as most important by practitioners. As for ease of use, we resort to Davis' questionnaire whose validity and reliability have been previously endorsed (e.g. [MPC01, AP98]).

Internal Validity is concerned with the conduct of the study. Here, the treatment is the use of *CustomDIFF* to address customization analysis by SPL engineers. We were specially careful to focus on SPL engineers who have at least one year of SPL expertise. Indeed, participants have on average, seven years of SPL experience and come from different backgrounds in SPL development. Hence, we believe participants to be representative of the target audience. As for the evaluation methodology, the questionnaire's understandability was improved by providing a running example that aimed to help participants on contextualizing the different questions.

External Validity tackles the representativeness of the study, and the ability to generalize the conclusions beyond the scope of the study itself. In this paper, validation was conducted with employees coming from a given SPL: Danfoss Drive SPL. Hence, customization practices reflect those of a single company, and hence, it rests to be seen whether *CustomDIFF* accounts for customization practices other than Danfoss'. For others to tap into this experience, Table 3.7 provides contextual details that might help others to extrapolate this experience to their owns. That said, it should be noted that *CustomDIFF* is an analysis tool and hence, it does not preclude the customization practice as such, in the sense of determining how to proceed during the pruning stage.

Context dimension	Characteristic	Value
Product	System Type	embedded systems
	Size (aprox.)	800 features & 20 products
	Maturity	10 years
	Programming Language	C++
Process	Customization practice	grown-and-prune
	Branching strategy	branch-and-unite
	Release cycle frequency	bi-monthly
Tools & Techniques	SPL approach	annotation-based
	SPL tool	*pure::variants*
People	Roles	domain engineers
	Experience	7-year average SPL experience
Organization	Model	Matrix organization
	Certification	SPLC-awarded hall of fame

Table 3.7: Danfoss Drive SPL contextual data along Petersen's facets [PW09].

Specially, we hypothesize that companies whose SPL is on a less mature level, where product customizations are more likely, would find CustomDIFF more useful.

For engineering science, Wieringa argues that "for theories to be useful in practice, they should give sufficient understanding of a sufficiently large class of cases, without having to be universal or complete" [WD15]. On the way to generalization, Wieringa introduces four strategies. In case-based generalization, we study individual cases, and generalize about components and mechanisms found in a case, by similarity. The assumption is that components are less varied than the cases they occur in [WD15]. This is the approach we follow. *CustomDIFF's* main contributions are pinpointing to the information needs, and the way to resolving these needs through DW and Alluvial diagrams. We believe this approach to be valid beyond variations on either the technological setting (e.g. data sources other than Git, annotation-based SPL frameworks other than *pure::variants*) or the process setting (e.g. different SPL release frequencies) or the organizational setting (e.g. a hierarchical structure instead of a matrix structure). *CustomDIFF's* main contributions are pinpointing to the information needs, and the way to resolve these needs through DW and Alluvial diagrams. We believe these contributions to be general enough to benefit SPL installations other than Danfoss.

3.9 Related work

This section frames *CustomDIFF* within related approaches on monitoring the application engineering process. Differences mainly stem from *what* is being monitored, *how* is being monitored, and *why* is being monitored. Table 3.8 outlines the outcome that also includes the type of SPL being targeted. Next, the comparison is arranged along the artefact being monitored ("the *what*").

Requirements. Here, *product engineers* are instructed to suggest eventual SPL

Work	**The subject of change (What)**	**Purpose (Why)**	**Change Detection Technique (How)**	**SPL type (Where)**
[HR10]	Requirements, Variability model	Uncover application needs	Continuous monitoring	na
[CKM+08]	Requirements	Foster feed-backing from application engineers	Story-based textual communication	na
[MBKM08]	Code	Promote cloned code to the SPL	Clone detection	Annotation
[LG15]	Variability model	Increase awareness of changes in products	Continuous monitoring	Composition
[PTS+16]	Code	Synchronize products among them	Continuous monitoring	Clone&own
[FLLE16]	Code	Synchronize products among them	Feature extraction	
CustomDIFF	**Code**	**Ease product customization analysis**	***Diff***	**Annotation**

Table 3.8: Related work on monitoring the application engineering process.

requirements to domain engineers. In Carbon et al. [CKM+08], and based on their interaction with customers, product engineers resort to *reuse stories* to directly communicate changes in SPL requirements to domain engineers. This approach adapts the agile practice "planning game" to SPLs [CKM+08]. In a similar vein, Heider et al. also advocate for SPL requirements to be fed from requirements risen during application engineering [HR10]. Unlike Carbon et al, Heider et al. do not require explicit intervention of domain engineers, but rather, application engineering is being monitored at the requirement level. To this end, authors introduce EvoKing, a tool that provides SPL engineers an overview on new requirements that have arisen on product level. Domain engineers can afterwards decide about each requirement to be implemented at the product level or SPL level.

Compared with EvoKing, *CustomDIFF* faces a similar problem but tracking is achieved at the code level by inspecting the SPL's Git repository. It could be argued that monitoring code rather than requirements makes *CustomDIFF* more "evidence-based" in the sense that what is being tracked is code that has already been delivered to customers. Drawing the attention of busy domain engineers would require not just grasping the product requirements but proving that the new functionality is being coded, tested and delivered to customers. What might well lie behind these different focuses is a distinct way of managing product engineering, i.e. whether product engineers are free to promptly account for their customer requirements by moving directly into code, or rather, customer requirements might need first some additional approval by domain engineers. On the downside, and unlike EvoKing, monitoring product engineering at the code level requires of additional mechanisms that abstract away from code to most abstract terms such as feature and product. This is being one of the endeavors of *CustomDIFF*.

Variability models. Here, *product engineers* can add features to the variability model if exiting functionality is not enough to fulfill customer requirements. Broadly, the variability model can be collaboratively edited, and the challenge is how to make all contributors aware of the change. To this end, Lettner et al. [LG15] introduce the notion of "features feeds". Domain and application engineers can *subscribe* to the variability model elements, i.e. configuration units, features and variation points (elements in the Common Variability Language [HMO+08]). Say a product engineer needs to add a new feature to a product, and hence, she adds a new feature to the configuration unit *CU1*. Engineers (both domain and application ones) subscribed to *CU1* will be notified. Next, when the new feature is implemented, product engineers can *propose* their implementation to be promoted as reusable, and if so, other engineers can incorporate it into their developments.

Compared with Lettner et al., *CustomDIFF* is less ambitious in the sense that our aim stops at detecting the change (i.e. the customization effort) but it does not elaborate on what should be the implications of such analysis. So far, *CustomDIFF's* main scenario is for domain engineers to schedule next SPL release. However, product engineers might also benefit from gazing at what other mates are customizing for the features of interest. For instance, a bug fix introduced in a given product might be promptly and directly incorporated into other products, without waiting for this fix to be promoted into the core-asset. Nevertheless, *CustomDIFF* does not preclude the actions derived from customization analysis.

The source code. Mende et al. [MBKM08] tackles clone detection among product customizations. Authors use clone detection techniques to identify similar functions among the derived products, that can later be re-engineered back to the SPL. Their approach uses the Levenshtein distance to measure the similarity between products' functions. They also propose metrics that aggregate the similarity at the architectural level to sustain the need for the pruning phase. For *clone&own* SPLs, Pfofe et al. [PTS+16] address product sync. For a given product change, an Eclipse plugin facilitates this change to be propagated to other feature-sharing products using a 3-way merge.

Compared with these works, our approach differs in the goal. We also work at the code level but our aim is not to detect clones nor comparing one implantation with another. Our analysis stops at detecting the modified LOCs as a measure of the customization effort. By contrast, we strive to provide abstraction and visualization mechanisms that permit engineers to conduct customization analysis at levels other than code.

3.10 Conclusion

Timely SPL evolution might require changes to be first conducted into products (grow), and next, be promoted into core-assets (prune). This grow-and-prune cycle might need to assess the extent and quality of "the growth" to better conduct the pruning. That is, grow-and-prune might need customization analysis. The main contribution of this paper is to introduce a DW approach to analyze this customization process. In this setting, we conducted a survey among Danfoss engineers to identify information needs.

Next, we resorted to Dimensional Modeling to tackle these information needs using the modified LOCs as *facts*. Finally, we proposed the use of Alluvial diagrams as a visualization mean. This approach is fleshed out in *CustomDIFF*, a DW tool that uses *git* as the operational system, and *pure::variants* as the SPL framework. The approach has been motivated and evaluated by Danfoss Drives SPL engineers. Primary evaluations reveal promising results on *CustomDIFF*'s usefulness for customization analysis.

Main limitations of the approach are the that ETL process depends on the variability approach, the VCS, and the branching model used. So far the approach works for annotation-based SPLs, *git* VCS, and branching models that keep the development of core-assets and products in separate branches. Another limitation of the approach, is that the approach focuses on the code assets and left aside other type of artifacts (e.g. models, documents). Future lines include extending the approach for other VCS and branching models, and other asset types.

Additionally, we would like to further evaluate *CustomDIFF* in different companies to measure its effectiveness along two parameters: the SPL maturity (which might impact the customization effort) and the SPL size (the larger the number of core-asset and products, the more compelling the need for abstract visualizations). In addition, we have so far focused just on two dimensions: "the what" and "the where". It would be of interest to study how to supplement *Git* data with data coming from other sources to collect information about products, customers and developers, and to see what other kind of analysis this additional sources would allow for. After all, DW are thought for integrating heterogenous data sources.

Another follow-on is to go beyond analysis into action. *CustomDIFF* is an analysis tool and hence, it does not preclude the customization practice as such, in the sense of determining how to proceed during the pruning stage. An interesting development would be using *CustomDIFF* within a DevOps framework where the customization effort (at its different abstraction levels) is tracked, and reactions can be attached to a certain customization-effort threshold being surpassed. Other scenarios include the use of *CustomDIFF* by product engineers to gaze what other mates are customizing. For instance, a feature enhancement introduced in a given product might be promptly and directly incorporated into other products, without waiting for this enhancement to be promoted as a core-asset. This opens new scenarios for SPL evolution where "longitudinal evolution" (between core-assets and products) might well co-exist with "traversal evolution" where products sharing same features might decide to incorporate enhancements from other products, and later on, be jointly pruned. The final aim is to find ways to alleviate the tension between the quality and re-use effectiveness required by domain engineers, and the time-to-market and customer pressure put on application engineers.

Chapter 4

Peering into peers

4.1 Overview

In the previous Chapter we addressed customization analysis, i.e. the practice by which engineers analyze product customizations, so that (a subset of these) are identified and promoted to the core-asset base. Customization analysis is the first step towards the pruning phase. Once the interesting functionalities are identified, these need to be propagated to the core-asset base. However, this practice might end up in the so-called "merge problem" (a.k.a. integration hell, or merge hell).

In this Chapter [1], we look at how to lessen the "merge problem" in SPLs. We advocate for making application engineers aware of potential coordination problems right during coding, rather than deferring it till merging time. To this end, we introduce the practice of *code peering*, i.e. the practice whereby product engineers inspect and compare other products' code with their own code, and if interested, merge the other product's code into his/her own product. We discuss four design principles that drive how code peering can be introduced for SPL development. As a proof-of-concept we developed *PeeringHub*, a tool tool that supports code peering through: (1) a Chrome extension that enhances GitHub with *peering bars* that provide brief information about what features are other peers changing, (2) a DW solution (similar to the previous Chapter) that provides alluvial-based high-level visualizations indicating the features available for *code peering*, and (3) *feature-based* 3-way comparisons so that product engineers can analyze how a given product is changing the code of a given feature w.r.t its own. A 13-minutes video showcasing *PeeringHub* is available at https://tinyurl.com/y7pe79h4.

Next Section provides the problem definition.

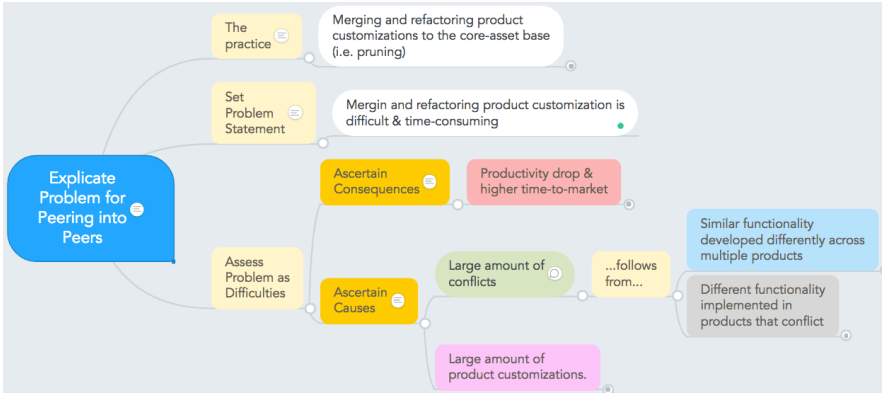

Figure 4.1: Mind map depicting the root-cause analysis for peering into peers. Interact with it online at https://tinyurl.com/y9fqucpe.

4.2 Problem definition

Following the *grow-and-prune* model, product customization (i.e the growth) needs to be cleaned up by merging and refactoring (i.e. pruning) [FV03]. Implementation wise, this is supported through Version Control Systems (VCSs) [WS02a]. VCSs support parallel development of software by maintaining a line of development (the *master* or *trunk*) with branches off this. For SPLs, the *master* holds the core assets while branches can stand for SPL products [FSK+16]. Product branches help to address product specifics in a secluded setting: developers can add commits to their local repository (*grow*) and completely forget about companion product developments till they are (fully or partially) merged back to the *master* (*prune*). At this time, however, resolving integration issues might be too demanding [mer]. Living on their own, products might diverge too much from each other, a known issue when communication channels are poor [DSB05, TMMK11]. Due to this issue the following problem arises: **merging and refactoring product customizations is difficult and time-consuming**.

Refer to Figure 4.1, which depicts the problem definition as a mind map, and outlines the causes and consequences of the problem. Refer to Chapter 1 for a detailed description on the root-cause analysis of the problem (i.e. cause and consequences of the problem). The reader is encouraged to interact with the mind map at https://tinyurl.com/y9fqucpe. The nodes can be unfolded to uncover the supporting evidences for each of the claims.

To alleviate this situation, we propose to make application engineers aware of potential coordination problems right during coding rather than deferring it till merging time. This idea might seem counter-intuitive. Product branches are all about speeding up customer-demand satisfaction. By caring about posterior merging, product engineers might delay this satisfaction. This begs the question whether it is worth

[1]The content of this Chapter has been accepted for publication in the International Conference of Software Product Lines (SPLC'18).

diverting developers' attention for the sake of later merging. We elaborate this issue with the help of the theory of Attention Investment [Bla02]. This theory is based on the premise that most decisions to start programming activities are based on an implicit cost-benefit analysis. In our setting, the cost is that of somehow tuning your code to that of your peers. But there exists also important benefits: early reuse. After all, products are all generated from the very same set of features, and hence, they share most of their code. If code is changed for a product's feature (e.g. bug fixing), then other products reusing that feature might be interested in the change. This scenario raises different questions. How can application engineering teams be aware of what others are doing without compromising their main duty (i.e. product development)? Which changes from other peers should they pay attention to?

In addressing these questions, this paper makes the following contributions:

1. a description of the roles and interactions that intermingle in a *grow-and-prune* approach to SPLs, motivated by the Danfoss case.

2. a characterization of the merge problem and how it differs from the merge problem that also appears in traditional single-system development. We propose a new practice, *code peering*, as a possible way to alleviate it. This begs the question whether it is worth diverting product developers' attention for the sake of making easier the subsequent pruning by domain engineers. Using the theory of Attention Investment [Bla02] as a narrative, we introduce four design principles that drive how code peering can be introduced for SPL development.

3. a realization of these principles using GitHub as the VCSs, and *pure::variants* as the SPL framework. As a *proof-of-concept* we developed *PeeringHub*, a tool that supports code peering through: (1) a Chrome extension that enhances GitHub with *peering bars* that provide brief information about what features are other peers changing, (2) a web-based application that provides alluvial-based high-level visualizations indicating the features available for *code peering*, and (3) *feature-based* 3-way comparisons so that product engineers can analyze how a given product is changing the code of a given feature w.r.t its own.

We start by characterizing the *grow* phase.

4.3 Characterizing the grow phase

This Section outlines the grow process, based on the Danfoss case. As a running example, consider the *WeatherStationSPL*, an SPL for building web-based applications for weather stations[2]. Figure 4.2 depicts a certain stage in the *WeatherStationSPL* evolution. So far, this SPL has undergone a single core-asset baseline release at the *master* branch, i.e *Baseline-v1.0,* that holds seven features realized through 30 code assets. Let us consider that customers urge to account for product specifics with no time to wait for the next core release. As a result, *Baseline-1.0* might be branch off into *productDonosti*, *productNewYork*, *productDenmark* and *productLondon*.

[2]This example is slightly different form the one presented in Chapter 3.

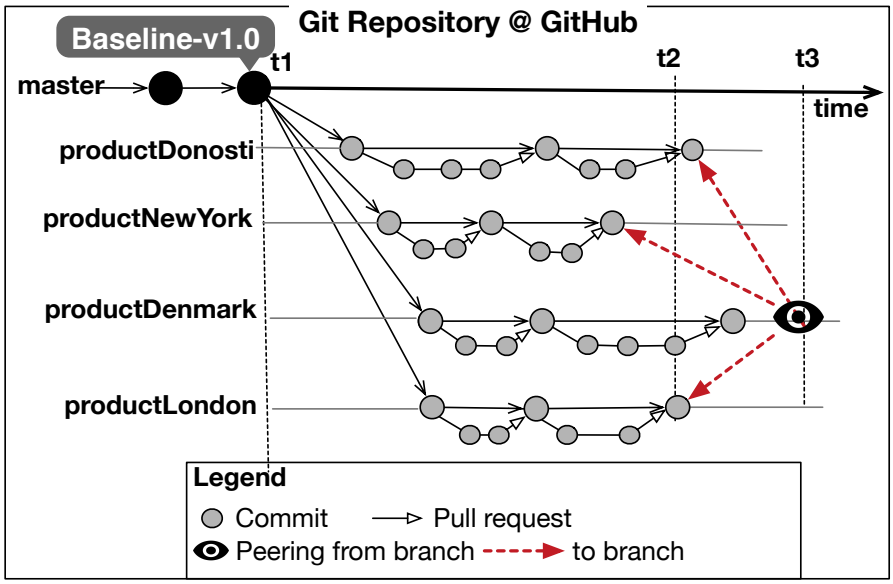

Figure 4.2: *WeatherStationSPL* branching model: the *master* branch holds the core assets baselines from where SPL products are branched off. At time *t3* *productDenmark* conducts code peering.

Figure 4.3 displays the main roles and interactions along the lines of the Danfoss experience. The interaction starts with the **Platform Release Manager** delivering a new release of the core assets (in the *master* branch using the **Version Control** agent). Next, the **Core Team**[3] branches off for each product where product configurations are tested out. The diagram highlights the two main triggers of the *grow* stage: bug fixing (verification) and product enhancements (validation). The former involves the **Quality Assurance** actor while validation results from User Acceptance Testing (UAT) with the **Customer**. Both, bugs and enhancements requests, are first communicated to the Core Team, which is the one that sets priorities and commands development. Both verification and validation activities are handled through the **Change Management** agent.

Figure 4.3 highlights the interactions with the **Version Control** agent (in bold): product branches are created by the **Core Team**, and elaborated upon by the **Software Developer**. Eventually, product branches are merged back into the *master*, and merge issues are resolved by the **Core Team** (not shown in the figure). Merge conflicts arise when **Software Developer**s were working on the same codebase. The likelihood of this situation very much depends on the size of the merge, which in turn, it is influenced by the time span from the last merge. For Danfoss, this timespan is two weeks.

[3]Members of Core Team are Release Manager, Product Manager , Project Manager (who interacts with developers and Platform Manager) and a person from the verification and validation team (QA) to understand on the schedules and testing.

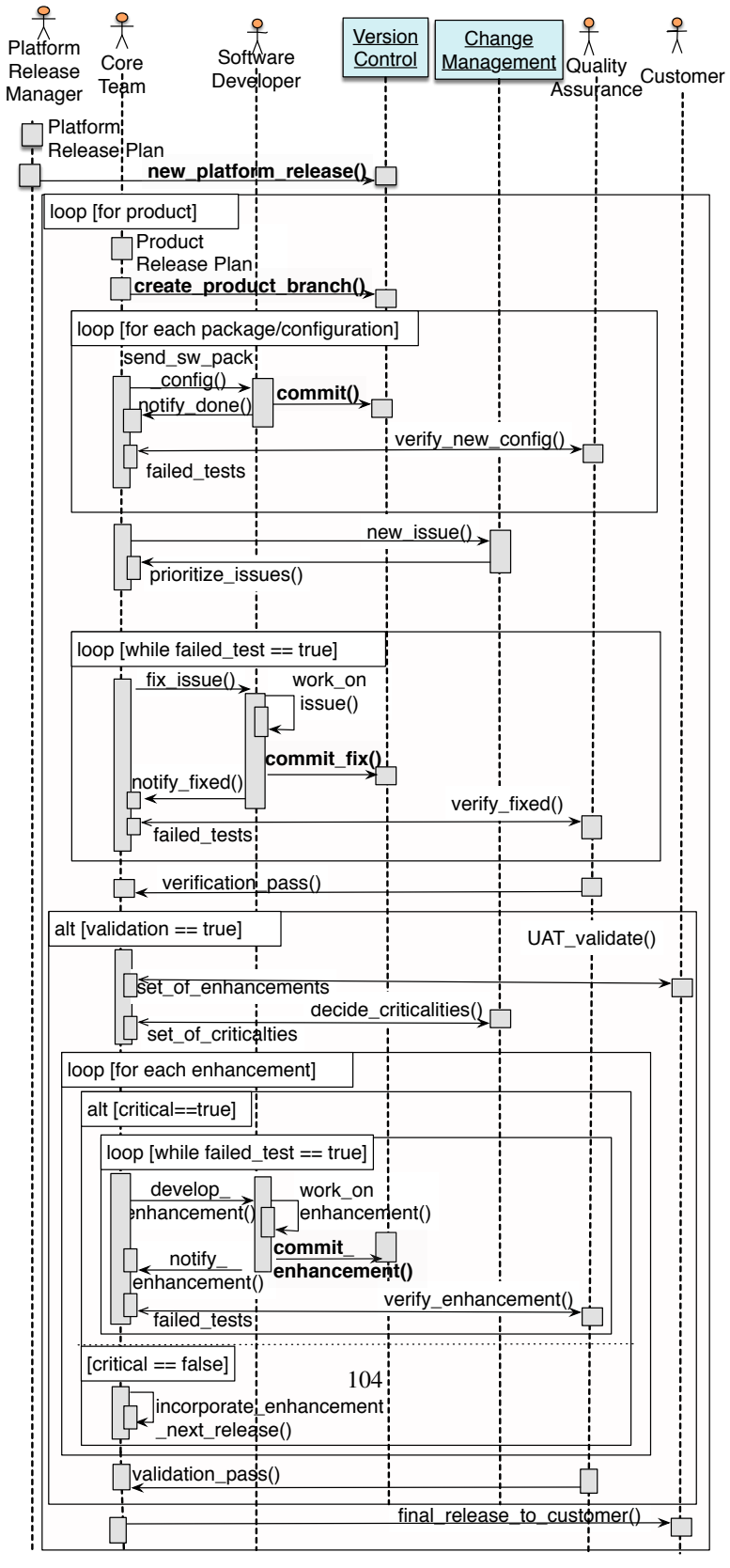

Figure 4.3: Sequence diagram depicting the *grow* stage.

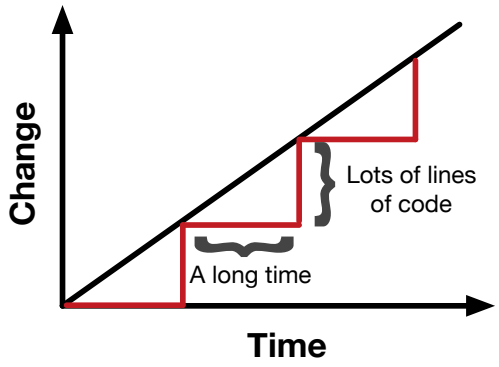

Figure 4.4: The merge problem illustrated: the *time* since the last merge and *the amount of changes* introduced since then, exacerbate the merge problem.

4.4 The merge problem

This section tackles the merge problem in an SPL setting. This problem is being studied for single-system development [Duv07]. This issue rises when long-living branches are merged back into the *master* branch. Here, the amount of code to be integrated might exceed the time it took to make the original changes, leading to the so-called "integration hell" [mer]. The likelihood of this situation very much depends on the size of the merge, which in turn, is influenced by the time span from the last merge. Figure 4.4 depicts this situation. The larger the span, the harder the integration. Developers should then aim at frequent merges, easy to be integrated with the base code.

However, in SPLs, product customization cannot be readily made available at the *master*. First, they need a proving time at the product realm before being promoted to core assets. Hence, if we cannot always reduce the merge granularity by frequent integration, we can alternatively attempt to make product engineers "merge-minded". That is, making application engineers aware of potential coordination problems right during product coding rather than deferring it till merging time. However, to what extent is diverting developers' attention worth, for the sake of the posterior merging? We elaborate this issue with the help of the theory of Attention Investment.

This theory is based on the premise that most decisions to start programming activities are made based on an implicit cost-based analysis [Bla02]. Specifically, the following parameters are introduced:

- Investment: the attention expended toward a potential reward, where the reward can either be external to the model (such as payment for services) or an attention investment pay-off. Back to merging, caring for integration during product development requires an extra effort of comparing your code with someone else's.
- Pay-off: the reduced future cost that will result from the way the user has chosen

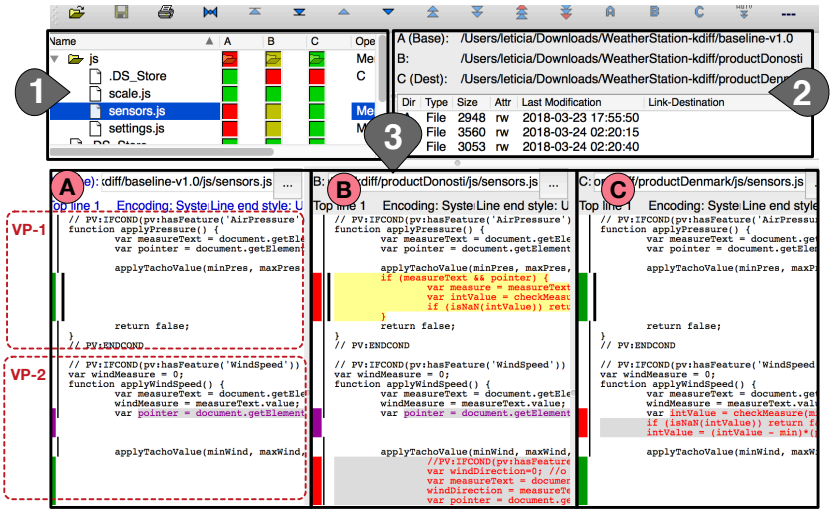

Figure 4.5: A 3-way comparison in *KDiff3* for *sensors.js*. The comparison involves three branches (see Figure 4.2): *Baseline-v1.0* (A), *productDonosti* (B) and *productDenmark* (C). Note how *sensors.js* is being changed in *productDonosti* for two variation points: VP-1 and VP-2.

to spend attention. Back to merging, the comparison effort might payoff in two main ways. First, promoting early reuse. If a feature's code is changed in product, then, this change would likely be of interest to other products that exhibit the same feature. Second, a developer who sees an opportunity to incrementally improve the quality of the codebase might somehow tune his/her code to facilitate late merging.

- Risk: the probability that no pay-off will result (specification failure), or that additional costs will be incurred (bugs). At this respect, it is being documented that the type of small but broad refactorings that can gradually improve a codebase—or stop it gradually degrading—are exactly the type of changes that often lead to a merge conflict [sem].

The hypothesis is that *providing developers with integrated support for product-branch comparison would promote early reuse and small refactoring improvements, on the search for easy merging back into the master branch.* Next Section characterizes this challenge which is referred to as "code peering".

4.5 Characterizing "code peering" in SPLs

Code peering is the process whereby developers look into someone else's code w.r.t. their own code. Specifically, developers look at sibling branches, i.e. branches that

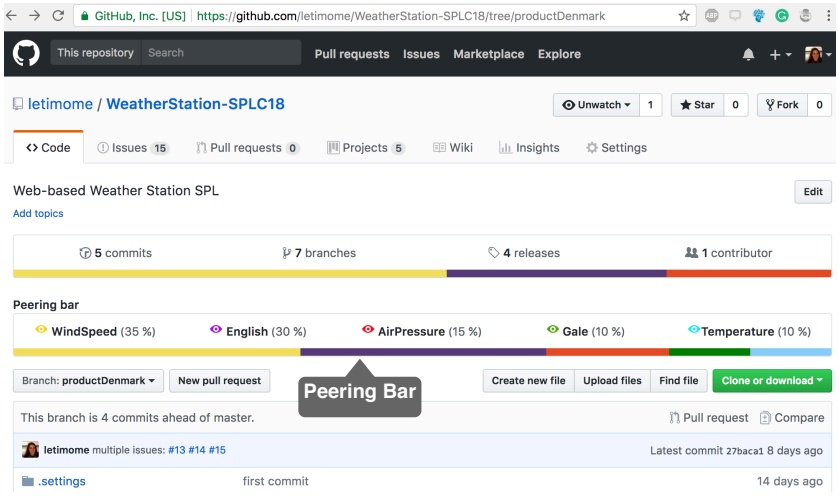

Figure 4.6: Product-branch display in *GitHub*. The inlayed peering bar hints customization endeavors i for the *productDenmark*'s features.

conduct customizations on products (i.e. the *observed* products) that share at least one feature with the product at hand (i.e. the *observer* product). Hence, code peering involves comparisons between the *observed* products and the *observer* product w.r.t. a common ancestor, i.e. the *master* branch. This requires a three-way file comparison: 3WAYDIFF (*base*, *observed*, *observer*). Figure 4.5 depicts how a 3WAYDIFF (*Baseline-v1.0*, *productDonosti*, *productDenmark)* looks like in *KDiff3*[4], a popular three-way comparison tool (for a larger list on three way comparison tools refer to [3wab]). The *base* (a.k.a "common ancestor") provides the reference point for conducting the comparison. Using the *base* as the reference, *KDiff3* compares (1) the *observed* vs. the *base*, (2) the *observer* vs. the *base*, and (3), the changes in both the *observed* and the *observer* to each other. For instance, Figure 4.5 highlights that *productDonosti's* version of the *sensors.js* file, holds an *if-clause* (yellow background) that is not present in *productDenmark*'s. Without the presence of the *base* (i.e. Baseline-v1.0), we would not know whether *productDenmark* did remove the *if-clause* from the *base*, or instead, it was *productDonosti* the one that included the *if-clause*. The comparison with *Baseline-v1*.0 shows this up.

KDiff3 is a powerful tool for code comparison in single-system development. Next, we frame code comparison within the *grow-and-prune* model. Here, code comparison is conducted *for* alleviating branch merging *within* an SPL setting. Next subsections elaborates on the implications in terms of design principles that might guide the introduction of code comparison in SPLs.

[4] `http://kdiff3.sourceforge.net/`

4.5.1 Code comparison *for* alleviating branch merging

Code peering encourages easy merging. This might be an ancillary activity from an AE perspective, as for them, product development comes first. Therefore, the attention capital available for code peering is limited. Implications are twofold. First, code peering should not interrupt product development. Second, if the peering effort goes beyond a certain threshold, the benefits might not payoff. On these premises, two design principles are introduced.

Seamless integration with VCSs. The need for code peering arises when conducting product development in VCS's branches. Hence, easy access from VCSs becomes paramount to promote "code peering" right at the place where the need emerges.

Respect focus. Along the theory of Attention Investment, the design should respect the fact that developers should control their own focus of attention. Product development should not be interrupted with merging concerns but developers should choose the *appropriate moment*. One might think that by letting developers decide, the appropriate moment for conducting code peering would never happen, as they would be too busy. Following the attention investment theory, we argue that making developers aware of early reuse mitigates this. Nevertheless, the development process should enforce developers to conduct code peering when developing, i.e. product developers should "look" for reuse opportunities in other peers just like they would do with domain assets.

4.5.2 Code comparison *within* an SPL setting

Feature-centricity. SPL development is basically feature-centric, i.e. most bug fixing and functional upgrades are conducted within a feature. Certainly, products can add brand-new functionality. However, our focus is on upgrades on existing features. This feature-centricity should percolate code comparison. That is, code comparison should be in terms of features, not just files or folder. Developers wonder "what features have been upgraded by my peers", and not "what files/folders have been updated". This requires a clear understanding of feature boundaries and an explicit feature-to-code mapping.

Abstraction. SPLs are reckoned to exhibit a large number of features and products. Hence, even if we limit our attention to the features of the product at hand, the number of sibling products (i.e. those with overlapping features) can still be quite large. Code-based comparison might not scale up. Conducting traditional *KDiff3-like* comparison for each file within each product would end up in hundreds (if not thousands) of *KDiff3* displays. This might put developers off. But even if they are dedicated enough, developers might overlook some interesting upgrades, hidden in the plethora of changes. Hence, mechanisms are needed that abstract away from raw code into higher-level visualizations.

4.6 PeeringHub: a peering utility for GitHub

This section realizes the aforementioned principles for *KDiff3* as the DIFF tool, *Github* as the VCSs, and *pure::variants* as the SPL framework [pur]. We first outline these tools, and next, address how they have been integrated and adjusted for "code peering".

Pure::variants. It is a framework for annotation-based SPLs. Figure 4.5-A shows a snippet of the core asset *sensors.js* at Baseline-v1.0. This snippet shows two variation points, i.e. VP-1 and VP-2. In *pure::variants*, a variation point starts with the opening directive *//PV:IFCOND,* and ends with a closing directive *//PV*:*ENDCOND* [5]. Core assets can be branched off for verification (corrective maintenance) or validation (perfective maintenance) purposes. Figure 4.5 highlights changes when comparing *productDonosti* (B) vs. *productDenmark* (C). Two variation points (i.e. feature expressions) are impacted: VP-1 and VP-2, that account for features *AirPressure* and *WindSpeed*, respectively.

GitHub[6]. It is the largest *git* repository hosting site [GVSZ14]. *Github* became "the social meeting point" for software developers working on the same repository. This "social activity" includes: branch comparison, pull-requests creation, conducting code-reviews, open/close issues, leave comments, see statistics and so on. This "social character" makes GitHub also a suitable place for code peering. Although GitHub has been made popular for open-source development, it is also popular for companies developing commercial software too [KDB+15b].

4.6.1 PeeringHub: code peering in GitHub

PeeringHub is a *Chrome* extension for *GitHub*. This extension enhances *GitHub* with three main utilities: peering bars, alluvial diagrams, and *KDiff3* enactors. *PeeringHub* has been designed along the aforementioned principles.

Seamless integration with GitHub. *PeeringHub* resorts to Web Augmentation [DA15] to enhance *GitHub* with code peering. This permits developers to keep using their URLs as usual. The new functionality is realized in terms of inlayed HTML elements (see next). Seamlessness is sought by using the same CSS classes and aesthetics of the hosting pages.

Respect focus. Code peering should not imply constantly popping up code differences. This would have been very annoying. Rather, developers should be able to consult about differences anytime they see appropriate. That said, giving full control to developers does not tell the whole story, because there is also an attention cost involved if the system simply waits for developers to proactively ask questions. *Peering bars* provide a middle way. Before delving into the bar itself, it is worth discussing where this bar is going to be integrated. The "respect focus" principle suggests the bar to be close to where the decision is to be taken. If code peering is conducted before

[5] These variation point patterns only hold for code files. For example, in XML and HTML files, variable elements are annotated in an attribute called *condition*.

[6] `http://github.com/`

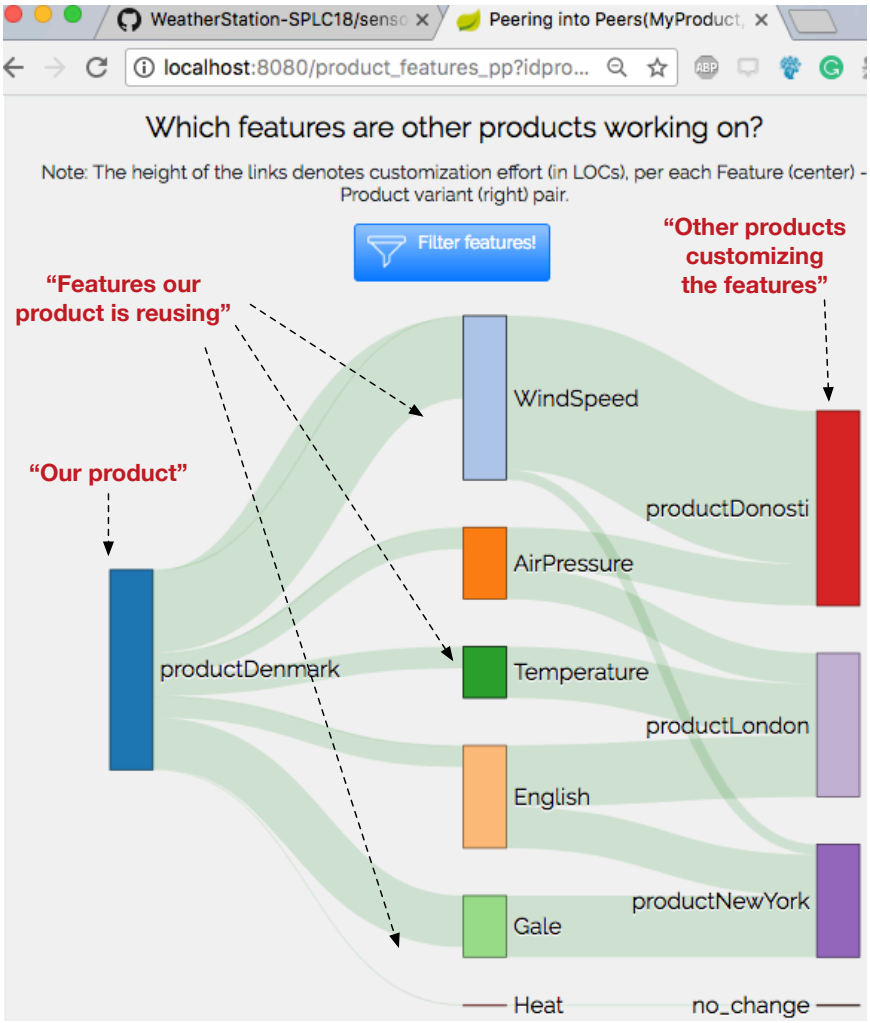

Figure 4.7: Alluvial diagrams reachable from peering bars. The display shows two flows (i.e. customization efforts): (1) from *productDenmark* into its features, and (2), from *productDenmark's* features to sibling SPL products.

initiating a development task, or at the end of the development task, this places the central repository (where the issue tracking system is) as the first option. Else, if code peering is conducted while in development, this places the IDE (e.g Eclipse) as the first option. We decided to integrate the peering bar as part of the GitHub interface, as a way to provide an IDE-indepent solution, since each developer (at least at Danfoss) can develop with his/her preferred IDE.

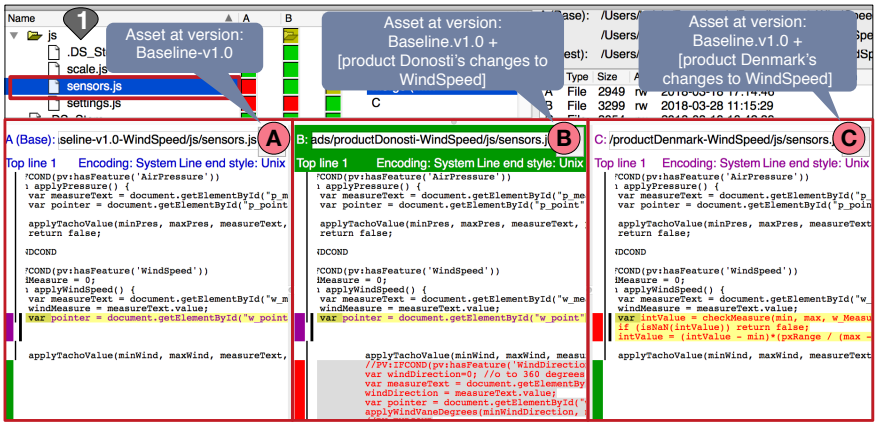

Figure 4.8: *KDiff3* enactment that results from clicking on the *(WindSpeed, productDonosti)* arch in Figure 4.7.

Peering bars mimic GitHub's language bars. Figure 4.6 shows the case for the *productDenmark* branch. To avoid distraction, the peering bar is initially collapsed: the existence of product customization is indicated but not the extent of this customization. Additional information requires for developers to proactively click on the bar. For our example, this will result in displaying the extent to which other products are altering *productDenmark's* features. Should a large number of features be involved, the display limits itself to those features that have received most attention, i.e. those features that have been subject to most customization in the *productDenmark* branch. Therefore, the peering bar displays changes dynamically as *productDenmark* is being customized. In this way, and without leaving *GitHub*, developers can assess whether it might be worth to zoom into the raw code or not. This bring us to the next principle.

Abstraction. *PeeringHub* gradually unveils the customization effort through three visualizations: bars, alluvial diagrams and finally, raw-code differences. A peering trail is built-in through hyperlinks that permit to move forward along these different visuals.

This trail starts at peering bars. These bars show aggregated feature-based customization efforts (see Figure 4.6). Click on these eye-shaped figures for these aggregates to be broken down through alluvial diagrams. These diagrams are a type of flow diagram originally developed to represent how multiple groups relate to one another across several variables [All]. Here, code upgrades are characterized along two variables: the product (where the upgrade is conducted) and the feature (that scopes the upgrade within a variation point). Figure 4.7 shows the case for our running example. The diagram depicts "the customization effort" that goes from the *observer* product (e.g. *productDenmark*) to the features, and next, from the features to the *observed* products (e.g. *productDonosti*, *productLondon* and *productNewYork*). This effort is measured in terms of the number of lines (LOCs) added/deleted (a.k.a the code churn), and it is reflected through the width of the flow arc.

Figure 4.9: Feature-based slicing for *diff(Baseline-v1.0.sensors.js, productDonosti.sensors.js)*. The diff-output (left) is broken down based on variation points (right). Each slice accounts for a patch function.

Looking at Figure 4.7, we can promptly appreciate how *productDenmark* is customizing the *WindSpeed* feature the most. Additionally, we can notice how both *productDonosti* and *productNewYork* also customize *WindSpeed*, being *productDonosti* the one with the largest customization effort (the arch flow stream to *WindSpeed* is the widest). Alluvial diagrams also help to promptly appreciate which variables are more clustered (fewer, wider arch flows) and which are more distributed (more, narrower arch flows). For instance, *productDonosti is* considerably more customized than the other two products, whereas the *Heat* feature has not been customized by any product at all. In this way, alluvial diagrams provide an abstract view of the customization effort. A flow arc (P, F) stands for the customization effort that product P conducts in feature F. The width of the arc denotes the amount of this effort. However, we do not yet see the concrete LOCs being added/deleted. This brings us to *KDiff3*.

Flow arcs account for enactors of 3WAYDIFF (*base*, *observed*, *observer*) comparisons. Specifically, when working on the P1 product branch, a flow arc (P2, F) holds an enactor to *KDiff3(base, P2, P1)*[7]. For instance, by clicking on

[7]This requires *KDiff3* to be locally installed, as well as, to grant permission to the protocol

the arc *(productDonosti, WindSpeed), KDiff3* will be launched in your desktop to show 3WAYDIFF (*baseline-v1.0*, *productDonosti*, *productDenmark*). Unfortunately, a straight invocation to *KDiff3* will highlight all the changes that both, *productDonosti* and *productDenmark,* have performed to the baseline. This means that changes to all the features will be highlighted. Hence, the feature of the flow arc "has been lost in translation". What is needed is a "feature-aware" 3-way diff, i.e., 3WAYDIFF (*baseline-v1.0*, *productDonosti*, *productDenmark*)[*WindSpeed*], so than only the changes to the feature *WindSpeed* are shown (as in Figure 4.8). This moves us to the next requirement.

Feature-centricity. Alluvial diagrams hold arcs *from* features *to* observed products. That is, arcs are anchored on features. Hence, differences need to be shown between the *observer* product and the *observed* product, but *for the feature at hand*. That is, the code should just focus on the feature being looked into. This means that we need to "build" versions of *productDonosti* and *productDenmark,* with only the changes that they introduced for the *WindSpeed* feature. So, that when *KDiff3* is launched only the changes to the feature *WindSpeed* are shown. To this end, *PeeringHub* proceeds in three steps.

- first, it conducts *diff(base, observer)* and *diff(base, observed)*. Figure 4.9 (left) shows the diff-output (a.k.a *patch*) for *diff(Baseline-v1.0.sensors.js, productDonosti.sensors.js)*[8],
- second, it slices the diff-outputs in terms of features. Figure 4.9 (right) shows the "featured" patches for the diff-output sample (left). This results in a patch for each variation point: *patchVP-1*, *patchVP-2* and *patchVP-3*. *VP-1* impacts *AirPressure*. *VP-2* impacts *WindSpeed*. And most interestingly, *VP-3* impacts *WindDirection* but also *WindSpeed* since it is nested within *VP-2*,
- third, patches behave as functions, i.e. they list the code lines being added or deleted. If you apply a patch to a file, it returns the file with the patch directives (addition/deletion) being performed. Therefore, *applyPatch(Baseline-v1.0, patchVP-3 ● patchVP-2)* returns a version of *productDonosti* with only the changes performed to the *WindSpeed* feature since derived from the baseline.
- forth, *KDiff3* is launched with the "featured" versions of both the *observed* and the *observer*. The outcomes are shown shown in Figure 4.8(B) and (C) respectively, after *PeeringHub* conducts these operations transparently. Unlike Figure 4.5, now the outcome limits itself to changes that impact *WindSpeed* alone. In this way, 3-way comparison is adjusted to SPL specifics, i.e. feature centricity.

kdiff:// so that *KDiff3* can be launched from the browser. This can be achieved by running a lightweight script, such as the following for Mac OS X `https://gist.github.com/letimome/4f8bd099c74f5226b98b09976f6812b7`.

[8]Due to space limitations Figure 4.9 does not show the *diff(Baseline-v1.0., productDenmark)*

Item for usefulness:	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**Avg.**
Using *PeeringHub* would enable me to accomplish *code peering* tasks more quickly	7	7	7	6	6	7	6.66
Using *PeeringHub* would increase my productivity on *code peering*	6	7	6	5	7	7	6.33
Using *PeeringHub* would enhance my effectiveness on the *code peering* job	6	6	6	5	6	7	6
Using *PeeringHub* would make it easier to do my job *w.r.t. code peering*	7	6	6	6	5	7	6
I would find *PeeringHub* useful for *code peering*	7	7	7	7	6	7	6.83
I would find *PeeringHub* useful in my job	7	7	4	7	7	6	6.33
Total	6.66	6.66	6	6	6.16	6.83	**6.36**
Items for ease of use:	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**Avg.**
Learning to operate with *PeeringHub* would be easy for me	7	7	7	7	6	7	6.83
I would find it easy to get *PeeringHub* to do what I want it to do	7	7	6	6	5	7	6.33
My interaction with *PeeringHub* would be clear and understandable	5	7	6	6	7	7	6.33
I would find *PeeringHub* to be flexible to interact with	7	6	5	6	6	6	6
It would be easy for me to become skillful at using *PeeringHub*	7	7	6	7	7	7	6.83
I would find *PeeringHub* easy to use	6	7	6	7	7	7	6.66
Total	6.5	6.83	6	6.5	6.3	6.8	**6.48**

Table 4.1: *PeeringHub* perceived *usefulness* and *ease of use* based on Davis' template.

4.7 Evaluation

This section predicts the acceptability of *PeeringHub* based on the Technology Acceptance Model (TAM) [Dav89]. TAM proposes that the readiness of a user to use (or not to use) a new technology is determined by her attitude towards the technology. This attitude is influenced by two beliefs which are *perceived usefulness* and *perceived ease of use*. Perceived usefulness is defined as "the degree to which a person believes that using a particular technology would enhance his or her job performance" [Dav89]. On the other hand, ease of use refers to "the degree to which a person believes that using a particular system would be free of effort" [Dav89]. Therefore, we aim at *analyzing the use of PeeringHub for the purpose of evaluating its usefulness and ease of use with respect to conducting code peering from the point of view of product developers in the*

context of annotation-based SPLs.

Six participants took part on the evaluation. Participants were introduced to the tool, by a short demo available at `https://vimeo.com/262269218`. Afterwards, they were introduced to the *WeatherStationSPL*. They were asked to conduct code peering using *PeeringHub*, as if they were engineers working for the *productDenmark* product. Specifically, the following tasks were proposed:

- *Task 1: Which products are changing features also used by your product (i.e. productDenmark)?*
- *Task 2: Which products are customizing the WindSpeed feature? List them. Which is the product that is customizing WindSpeed the most?*
- *Task3: For the previously identify products, list lines of code being added to the WindSpeed feature.*

During the session, a researcher was observing participants' interactions with the tool. Next, an on-line questionnaire was delivered to assess *usefulness* and *ease of use.* Table 4.1 gathers the results where agreement with statements is rated along a LIKERT scale that ranges from 1 ("Strongly disagree") to 7 ("Strongly agree"). Davis' template was used for evaluating both *usefulness* and *ease of use*.

Participants rated *PeeringHub* with an average of *usefulness* and *ease of use* of 6.36 and 6.48, respectively. Although results are rather encouraging, they should be interpreted with caution, as some threats to validity need to be considered. Internal validity is concerned with the conduct of the study. Here, the treatment is the use of *PeeringHub* to address code peering. We cannot claim *PeeringHub* to be tested in a real scenario, as it was conducted by SPL researchers with a sample SPL. Also, this work is based on insights from the Danfoss setting. This setting, i.e. the SPL size, the number of developers or the SPL maturity, might change for other companies that might rise issues not addressed here. Finally, external validity tackles the representativeness of the study, and the ability to generalize the conclusions beyond the scope of the study itself. At this respect, we believe our insights can be of interest to SPLs other than Danfoss'. Wherever product branches are permitted, the risk of difficult merges shows up. Code peering, and *PeeringHub*, might alleviate this scenario. While this evaluation provides some initial evidence that the proposed tool could be useful and easy to use for code peering, it is only a starting point for a more large-scale evaluation. We still need to evaluate whether conducting code peering with *PeeringHub* can alleviate the merge problem.

4.8 Related work

PeeringHub monitors the application-engineering process. Differences with other works mainly stem from *what* is being monitored, *how* is being monitored, and *why* is being monitored. Table 4.2 outlines the outcome that also includes the type of SPL being targeted. Next, the comparison is arranged along the *"what"*, i.e. the artefact being monitored.

Ref.	The subject of change (*what*)	Purpose (*why*)	Change Detection Means (*how*)	SPL type
[CKM+08]	Story-based requirements	Feed-back	Asynchronous communication	na
[HR10]	Requirements, Variability model	Feed-back	Monitoring	na
[LG15]	Variability model	Increase awareness of changes in products	Monitoring	Composition
[MBKM08]	Clone code	Feed-back	Levenshtein distance	Annotation
[PTS+16]	Code	Product synchronization	Monitoring	Clone&own
[SSRS16]	Code	Update propagation	*Diff*	Annotation
[**?**]	Code	Identifying features in forks	*Diff*	Clone&own
PeeringHub	**Code**	**Alleviate the merge problem**	***Diff***	**Annotation**

Table 4.2: Related work on monitoring the application engineering process.

Requirements. Here, *product engineers* are instructed to suggest eventual SPL requirements to domain engineers (a kind of feed-back propagation). In Carbon et al. [CKM+08], product engineers resort to *reuse stories* to communicate changes in SPL requirements to domain engineers. This approach adapts the agile practice "planning game" to SPLs [CKM+08]. In a similar vein, Heider et al. also advocate for SPL requirements to be fed from requirements risen during application engineering [HR10]. Unlike Carbon et al, Heider et al. do not require explicit intervention of product engineers, but rather, application engineering is being transparently monitored at the requirement level. To this end, authors introduce *EvoKing*, a tool that monitors requirement-level activities by product engineers. Domain engineers can afterwards decide about each requirement being implemented at the product level or SPL level. This tools was later used in [LG15] through the notion of "features feeds". Domain and application engineers can *subscribe* to the variability model elements, i.e. configuration units, features and variation points (elements in the Common Variability Language [HMO+08]). Say a product engineer needs to add a new feature to a product, and hence, she adds a new feature to the configuration unit *CU1*. Engineers (both domain and application ones) subscribed to *CU1* will be notified. Next, when the new feature is implemented, product engineers can propose their implementation to be promoted as reusable, and if so, other engineers can incorporate it into their developments.

PeeringHub differs from the aforementioned approaches in all: the target audience (domain engineers vs. application engineers), the artefact being monitored (requirements vs. code) and the SPL stage (requirement analysis vs. code development). At this respect, *EvoKing* and *PeeringHub* complements each other: requirement feed-back can be conducted through *EvoKing;* next, assigned to different product-engineering teams whose efforts and synergies are later tracked through *PeeringHub*. Monitoring wise, *EvoKing* requires product engineers to explicitly subscribe to the features they are interested in. By contrast, *PeeringHub* resorts

to the heuristic of "subscribing" products to those features that are being more intensively updated from those exhibited by the product at hand. Though products might potentially exhibit a large number of features, the heuristic limits the focus to only those features being upgraded in a short turnover (two weeks for Danfoss), hence averting the scalability issue in the presence of large feature models.

Source code. Mende et al. [MBKM08] tackles clone detection among functions during product customizations. To this end, they resort to the Levenshtein distance to measure the similarity between clones. They also propose metrics that aggregate similarities at the architectural level to sustain the need for the pruning phase. For the growing phase, Schulze et al. [SSRS16] address update propagation in *Pure::Variants* where products are upgraded with newer versions of the core-assets. Similar to our approach, authors resort to a 3-way diff/merge. However, the compared commits are different. Given a product generated out of BASELINE-1, they are interested in upgrading it with a new release of core assets, e.g. BASELINE-2. Therefore, their 3-way diff looks like: 3WAYDIFF (BASELINE-1, BASELINE-2, PRODUCT). By contrast, our approach looks for differences between products generated out of the same baseline. That is, our 3-way diff looks like: 3WAYDIFF (BASELINE-1, PRODUC-1, PRODUCT-2).

For *clone&own* SPLs, Pfofe et al. [PTS+16] address change synchronization between cloned products. Their tool, an eclipse plugin called *VariantSync,* tracks changes as engineers conduct product development. Afterwards, developers need to tag these changes to feature expressions, and the tool aids engineers on automating propagating those changes to products sharing the same feature expression. Although thought for fork-based development in open-source projects, the work presented by Zhou et al. [**?**] comes close to ours. Notice that forking is a common approach in *clone&own* SPL (e.g. [RKBC12]). In open-source projects, forks (i.e. a clone of a whole repository) can be interested in what related forks are doing. Zhou et al. propose a tool that compares each fork with the Main repository from where forks were derived. Unlike annotated SPLs, no explicit feature-to-code mapping exists so authors need to rely on both concern location and dependency analyses in order to identify features in forks. Finally, *clone&own* SPLs are regarded as the "first step" towards a fully-integrated SPL approach [**?**, RCC13]. In this sense, Antkiewicz et al. [**?**] and Rubin et al. [RCC13], propose roadmaps as to iteratively transition from a *clone&own* setting towards a fully-integrated SPL platform.

Back to *PeeringHub*, our efforts are not so much about concern location (since changes happens within the scope of a variation point) but at integrating the practice of "peering" as part of product enginering process. This moves visualization and gradual unveiling to the forefront.

4.9 Conclusions

This chapter proposes code peering as a way to lessen the merge problem during the pruning of product customizations. Using the theory of Attention Investment as a narrative, we introduce four design principles that drive how code peering can be introduced in SPLs. These principles are realized through *PeeringHub*, a prototype

composed by: (1) a Chrome extension that enhances Github with *Peering bars*, (2) a web-based application that visualizes code peering by means of alluvial diagrams, and (3) feature based 3-way comparisons.

While we got some initial evidences that the proposed tool could be useful and easy to use for code peering, a more large-scale evaluation should be conducted, that can shed some light as whether conducting code peering (with *PeeringHub)* can alleviate the merge problem.

Next follow-on activity is to measure *PeeringHub* effectiveness along two parameters: the SPL maturity (the less mature, the larger the need for code peering) and the SPL size (the larger the number of core asset and products, the more compelling the need for abstract visualizations). The impact of code peering can be measured not only in terms of facilitating branch merging, but also changing how product engineering is conducted. For instance, a feature enhancement (e.g. a bug fix) introduced in a given product might be promptly and directly incorporated into other products, without waiting for this enhancement to be promoted as a core asset. This opens new scenarios for SPL evolution where "longitudinal evolution" (between core assets and products) might well co-exist with "traversal evolution" where products sharing features might decide to incorporate enhancements from other products, and later on, be jointly pruned. The final aim is to find ways to lessen the tension between the quality and re-use effectiveness required by domain engineers, and the time-to-market and customer pressure put on application engineers.

Chapter 5

Synchronizing core-assets and products

5.1 Overview

In Chapter 4 we addressed the merge problem that arises during the pruning of product customization. Herein, we proposed code peering, i.e. a practice that promotes early reuse across products during the grow phase, with the aim of reducing the merge problem implications.

This Chapter[1], no longer focuses on preventive measures for the issues the pruning might await. Once the interesting functionalities are identified, these need to be propagated to the core-asset base, which after a successful integration, will eventually be delivered to the already existing products. This introduces two sync paths: *update propagation* (from DE to AE) and *feedback propagation* (from AE to DE).

We look at how to support these sync paths using traditional Version Control Systems (VCSs) constructs (i.e. merge, branch, fork and pull). In this way, synchronization mismatches can be resolved *à la VCS*, i.e. highlighting difference between distinct versions of the same artifact. However, this results in a conceptual gap between how propagations are conceived (i.e. update, feedback) and how propagation are realized (i.e. merge, branch, etc). To close this gap, we propose to enhance existing VCSs with SPL sync paths as first-class operations. As a proof-of-concept, we use Web Augmentation techniques to extend *GitHub's* Web pages with this extra functionality. This ends up in *GitLine*, a browser extension for Firefox that extends *GitHub* with sync operations for SPLs. Through a single click, product engineers can now (1) generate product repositories, (2) update products with newer feature versions, and (3), feedback product customizations amenable to be upgraded as core-assets. A 8-minute video showcasing *GitLine* is available at `https://vimeo.com/145403689`

This Chapter requires from the reader a basic understanding on VCS basic operations, and branching strategies. The appendix C provides the reader with a brief

[1]The content of this Chapter has been previously published in [MD15].

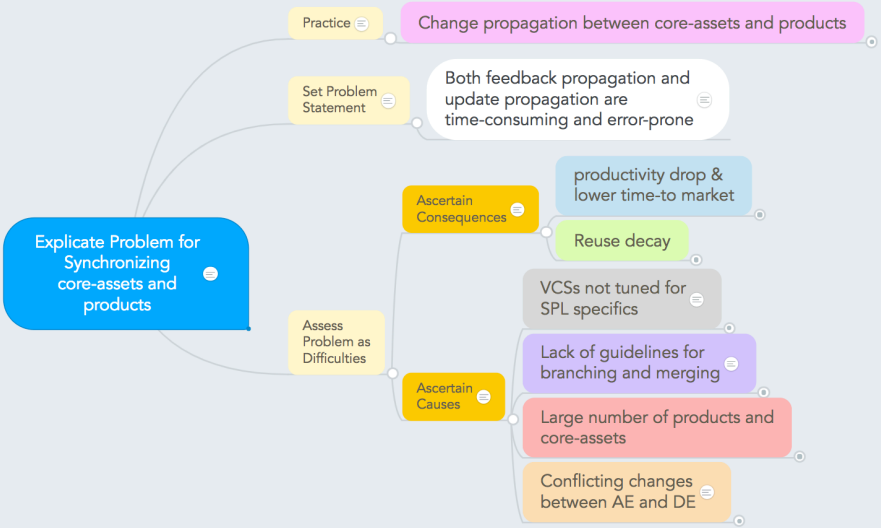

Figure 5.1: Depicting the problem definition for propagating changes between core-assets and products with a mind map. Interact with it online at https://tinyurl.com/ya777m2x.

on *git* VCS, its basic operations, and points to popular branching models for single-system development.

Next Section provides the problem definition.

5.2 Problem definition

Enacting the pruning requires propagating changes between core-assets and products, so that both parties are synchronized. This introduces two **sync paths**: the update path (from DE to AE), and the feedback path (from AE to DE) [Kru03]:

- Update paths serve two scenarios: configuration repair (synchronize products configuration when variability model changes) [BM14] & product upgrade (where latest versions of reusable assets are propagated to products) [Kru03]. In the latter case, for every product derived from the original core-asset, an update operation is required. If products have customized the core-asset then, the update operation may require a manual merge for each product [Kru03]. When to conduct the upgrade differs significantly for the different products in the SPL. While some tend to upgrade rather quickly, some do not upgrade for a long time, even when not close to the product's release [JB09].
- Feedback paths serve two scenarios: extending the scope of the product line to emerging application engineering requirements [Kru03], as well as, incorporating bug-fixes resolved in products [FSK+16]. The integration of the

feedback would result in changes to a set of core-assets, which may require updates to be applied to all the products that reuse them [Kru03].

In order to preserve a correct, complete and consistent synchronization between core-assets and products, Software Configuration Management (SCM) for SPL development needs also to account for propagations. SCM is the discipline that enables engineers to keep control and track software changes (i.e. evolution). Some equate SCM to VCS tools. However, beyond configuration management tools, policies and procedures are needed to guide developers in how to control and manage the evolution of the core-assets and products[McG07]. In a nutshell, SCM relies on both (1) **tools** to track changes to software assets, i.e. VCS, as well as, on (2) **policies** for engineers that establish when and how to commit code, and policies for branching and merging. If an organization chooses tools and practices separately, their use may conflict, resulting in failure to carry out the practices (e.g. SPL development and change propagation) properly [CN01a].

However, traditional VCSs tools are mainly thought for single-product development. State-of-the-art VCSs such as *Git/GitHub,* provide the basics but fall short in supporting sync paths between core-assets and products. All *Git/GitHub* offers is the *fork/branch* mechanism. However, forking/branching is not how products are derived. Likewise, GitHub's *pull request* / *merge* is also thought for synchronizing a whole repository/branch, hence lacking a more piecemeal synchronization, i.e. only a subset of features or core-assets. Due to this issue the following problem arises: **both feedback propagation and update propagation are time-consuming and error-prone.**

Refer to Figure 5.1, which depicts the problem definition as a mind map, and outlines the causes and consequences of the problem. Refer to Chapter 1 for a detailed description on the root-cause analysis of the problem (i.e. cause and consequences of the problem). The reader is encouraged to interact with the mind map at https://tinyurl.com/ya777m2x. The nodes can be unfolded to uncover the supporting evidences for each of the claims.

This complexity calls for Version Control Systems (VCSs), accompanied by branch and merge policies, not only to assist in managing the large number of SPL artifacts, but also to help in synchronizing the AE and the DE realms. This involves: (1) a model of what a *CoreAsset* repository looks like (a.k.a. **branching model**), (2) a model of what a *Product* repository looks like, and (3), a set of operations to keep both models in sync. In this setting, this work's main contributions rest on:

1. a repository architecture, which distinguishes between the *CoreAsset* repository, where domain engineering takes place, and *Product* repositories, where product engineering occurs. This provides the data structure branching model in which sync actions operate (Section 5.6).

2. the operational semantics for sync actions. Synchronization happens upon artifact versions. The previous branching model permits sync operations to be expressed in terms of basic VCS constructs. This in turn implies that eventual mismatches that rise during synchronization are resolved *à la VCS*, i.e. highlighting *diff*-erence between distinct versions of the same artifact

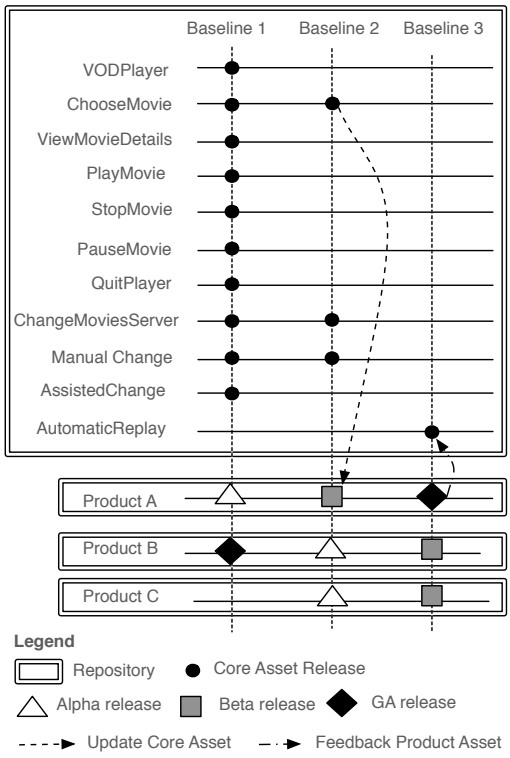

Figure 5.2: The SPL synchronization challenge (adapted from [KC13])

(traditionally, using the *diff* option in VCSs). Therefore, we do not aim at automatic sync. Our aim is much more humble: tap into VCS popular mechanisms for SPL engineers to achieve sync in a way similar to what they do for single products (Section 5). However, this results in a conceptual gap between how sync paths are conceived, and how they are realized down into branching and merging. To close this gap, we propose leveraging VCSs with SPL sync operations.

3. *GitLine*, a browser extension for *GitHub* that accounts for the above-mentioned sync operations (subsections 5.6.1.1, 5.6.2.1 and 5.6.3.1). Through a single click, product engineers can now (1) generate product repositories along a certain configuration, (2) update propagations of newer core-asset versions, or (3), feedback propagation of product customizations.

The next Section illustrates the synchronization challenge.

Core-asset ID	Core-asset Name	Core-asset Description
CA1	VODPlayer	Provides the PL architecture and basic functionality to run the player
CA2	ChooseMovie	Allow users to view the list of available movie and select one
CA3	ViewMovie Details	Allow users to see a movie details: director, title and actors
CA4	PlayMovie	Allow users to start playing the movie the have selected
CA5	StopMovie	Allow users to stop the movie they are currently watching
CA6	PauseMovie	Allow users to pause the movie they are currently watching
CA7	Quit	Allow users to quit from the player
CA8	ChangeMovies Server	Allow users to change the server they are connected to
CA9	ManualChange	A manual-like approach to change the server users are connected to
CA10	AssistedChange	Load a list of servers to allow users to select the one to connect to

Table 5.1: *VODPlayer-PL* core-assets.

5.3 Product derivation: illustrating the challenge

We stick to the generic process for product derivation described in [DSB05]. Deelstra et al. distinguish between the initial and the iteration phase. In the initial phase, a first configuration is created from the core-assets. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements. Unlike Configurable Product Lines (CPLs) where product derivation is limited to the configuration expression, SPLs do not achieve such degree of reuse effectiveness, and require core-assets to be customized during product derivation. This makes SPLs more difficult to manage that CPLs since they might potentially involve a larger number of artifacts (not just core-assets, but product specific artifacts as well), handled by different teams, and following different life-cycles. This Section illustrates the complexities of product derivation through an example.

Consider *VODPlayer-PL,* a SPL for video playing software. *VODPlayer-PL* includes ten core-assets at its initial version (see Table 5.1), is implemented in Java using Feature-Oriented programming [ABKS13a, BSR03], for FeatureHouse composer [AKL13]. Products are derived from those core-assets in accordance with a feature diagram (not included here). Both core-assets and products are not standing still but evolve. And this introduces the challenge: synchronize the pace at which core-assets and products are released, considering that those artifacts might well be governed by different teams with distinct priorities. Figure 5.2 depicts this matter. core-assets are arranged down the left-hand side(e.g. *VODPlayer*, *ChooseMovie*). Each asset undergoes evolutionary change; its evolutionary trajectory extends to the right. The bottom shows the products in the SPL. Each product goes through various phases, such as alpha release, beta release, and General Available (GA) release. Across the top are several baselines. A baseline contains a set of assets, each at a given version, that work together and are used to build products. Besides re-use of core-assets, Figure 5.2 also highlights possible sync paths (depicted as dotted lines): upgrades of *ChooseMovie* are percolated to *ProductA* whereas a customization conducted for *ProductA* is promoted as the core-asset *AutomaticReplay*. The question is how to facilitate this process using existing VCSs. The appendix C provides the reader a brief on VCS and *git* basic

operations and popular branching models.

5.4 Proposals on VCSs for SPL development

VCSs are a cornerstone for distributed, collaborative development. SPLs promote collaborative development through reuse. Traditionally, collaborative development applies to different users working on the same piece of code. By contrast, SPLs set two realms (i.e. domain engineering & application engineering), where collaboration goes along the sync paths. The fact of being two separated realms makes it even more important to track who made which changes, and when they were made. Provenance of the contributions can turn key when, like in the SPL case, development might be distributed among different business units with their own budgets and responsibilities [Bos01].

VCSs are specifically designed to keep track of who did what. Broadly, VCSs support "revisions", i.e. a line of development (a.k.a *baseline* or *trunk*) with branches off of this. Disparate efforts are reunited by merging branches. In addition, repositories can be *forked* whereby a whole repository is cloned in a separated space. Unlike a branch, a fork is independent from the original repository. If the original repository is deleted, the fork remains. This space can be merged back through a *pull request*[2]. The *fork-&-pull model* reduces the amount of friction for new contributors. This makes this model popular among open source projects because it allows people to work independently without upfront coordination. Notice that VCSs do not dictate the file structure nor when to branch or merge. This is part of the branching model. Approaches to branching models very much depend on the dependencies to be preserved through the VCS.

Back to SPLs, approaches broadly distinguish two main ways to face SPL development: *clone&own* (departing from existing products) and *managed* (departing from reusable assets). Next paragraphs delve into VCS proposed solution for these two scenarios[3].

Cone&own approach Here, a new product is obtained through *clone&own* from existing products. Branching model wise, there are three alternative models:

branch-per-product-customer [Sta04]: a main branch holds the code shared by all products. Product variants come as branches off the main branch, one per customer, where customer-specific modifications are performed.

branch-per-product-functionality [ABKS13b]: there is one main branch per functionality that products may exhibit. Product variants are obtained by merging functionality branches.

[2] `https://help.github.com/articles/using-pull-requests/`

[3] The reader might notice that some of the below mentioned works were already introduced in the background Section 2.5.3.3. In this case, we provide a discussion with a focus on VCSs and underlaying branching models.

repository-per-product approach, where there is a repository for each product being developed. The difference with the aforementioned approaches is that each product resides within a separate repository, instead of branches. Within each product repository any branching model for *single-systems* can be used to develop the product (refer to Appendix C). Thanks to distributed VCSs, if desired, a *fork&pull* model can be leveraged to *clone & propagate changes* between different product repositories [RKBC12].

As the authors themselves recognize, *clone&own* approach might be suitable as there is no need for a complete upfront scoping process [Sta04]. However, it also introduces overhead as it scales, since they encourage the development of product variants and not reusable core-assets. Notice that in clone-based SPLs, propagation takes place at the level of products in the absence of a "proper reuse". Therefore, activities such as propagating changes between product clones and creating new products based on previous clones becomes difficult [DRB+13], as well as, repetitive tasks are conducted (some tasks need to be performed on each cloned copy). In this context, other works have addressed the issue of aiding synchronizing of clone&own products. Rubin et al. [RKBC12] and Antkiewicz et al. [AJB+14] propose conceptual operations and discuss VCS implications to manage the synchronization of clones. An industrial experience on managing clone-based SPLs is later conducted by Rubin et al. [RCC13]. Authors conclude that an efficient management of clones relies on not only improving the maintenance of existing clones, but also refactoring clones into an SPL infrastructure. From a technical perspective, McVoy [McV15] introduces new VCS operations suited for BitKeeper, which enables opportunistic reuse and synchronization at component-level.

Managed approach Here, a distinction is made between core-assets (thought for reuse) and products (thought for use). From the perspective of VCS repository structure, three approaches have been reported:

single repository. Here, core-assets and products are kept in the same repository. Traceability between core-assets and derived products is achieved through branching [GP06]. On the downside, branches hold both core-assets and products. Sharing the same space might be a problem if these different kinds of artifacts are handled by different teams along distinct life-cycles. Scalability might also be an issue. Here, Anastasopoulos [Ana13] presents a tool on top of Subversion, which keeps SPL artifacts identified (where in the VCS are core-assets and products located). Engineers can perform activities related to evolution control including propagation of changes. Update propagation is performed by AE over a single core-asset instance. Feedback propagation is conducted by DE over a single core-asset. The feedback gets first all the changes performed to that core-asset by all the products, and merged them into the core-asset. This seems inconvenient since it assumes that all the instances have changes that need to be promoted to DE. We (latter) argue a more cherry-picking approach for feedbacking changes. Anastasopulos does not discuss implications for underlaying branching model. Calefato et al. [CNLL15]

propose a branching model for *git* (adapts git-flow [Gitb]), which is best suited for SPLs at a "platform" reuse level (i.e. where reuse is only for assets common to all products). For core-asset development there is only one branch: the *release* branch of core-assets (*master*) which holds reusable components for products. For the development of each product: (1) there is a *product release* branch, which branches from the core-asset *release* branch, (2) an *integration* branch which branches from the *product's release* branch, and merges back to it for releasing new product versions, and (3) *feature* branches for parallel development of product specifics which branch from products' *integration* branch and merges back to it when finished. We found some limitations with this branching model. First, a single branch for core-asset development seems to fall short. At least an additional core-asset integration branch should exist to aid in core-assets parallel development. Second, *updates* from core-asset to products, come as a merge from core-asset release branch into a product's' release branch. This seems risky since product specific changes might conflict with a new core-asset update, which would yield an unstable state of the product into the *release* branch. We argue that these updates should first be integrated into products' integration branch, and then, if correct, merged to products' *release* branch. Feedbacks from products' *release* branches to core-assets *release* branch follows a similar risk. This branching model might also face some limitations upon a high number of products being developed.

detached repositories. Here, core-assets and products are kept into independent repositories. There is one core-asset repository that serves for core-asset development and multiple product repositories that serve for product development [SSRS16, HSB]. Hellebrand et al. [HSB] present two branching models in *git*, by adapting *git-flow* [Gitb] (refer to the appendix C Section C.3 for more details on *git-flow*). The core-asset repository branching model extends *git-flow* with *multiple release* branches for core-assets releases (e.g. releases for versions 1.x, 2.x) to support the maintenance of old released product versions. The product repository branching model extends *git-flow* with branches though for *synching* with the core-asset repository, i.e. *a generation* branch. A product is derived from a core-asset repository *release* branch (e.g. baseline 1.0) by generation. This implies that the product repository, no longer holds the core-asset instances, but it holds the post-compiled (transformed) source files. Therefore, only the resulting generated product is committed to product repositories' *generation* branch. As the repositories are detached, another tool rather than the VCS, needs to store the relationship between the pairs core-asset&product repositories. This is managed by the variant management tool, which is in charge on creating product repositories whenever a new product is created from the variant management tool, as well as, enacting the update propagations[SSRS16]. Authors approach does not support feedback propagation. Since, product repositories hold only the generated product, and hence, the product-specific changes modify the generated product (not the core-asset instances), a VCS merge from any product repository branch to the core-assets makes no sense, since the core-assets have variation points and

products have already resolved them. This, could have been solved if product repositories hold the core-assets instances, instead of the generated products. In this sense, product engineers could make changes to the core-assets instead to the product itself and feedback them to the core-asset repository. Product generation (transformation or composition) can be made after the product-specific changes were done. This is the approach we follow.

linked core-asset and product repositories. Here, core-assets and products are kept in different repositories, although they are tied-up through a *derivation trace* so that subsequent syncs can be conducted by means of branching and merging operations[TMN08]. Unlike Anastasopoulos, Thao et al. [TMN08] do not consider reusing existing VCSs. Instead, they build a home-made one, which is capable of establishing dependencies between products and core-assets. They support built in product derivation, where the product engineers select the set of features they want to reuse, and a new product repository is created. Unlike Hellebrand and Schulze et al., the product repository does not hold the composed product, but just the components belonging to the features that product engineers have selected. Whenever a product is derived, a new branch is automatically created in the core-asset repository. This branch references the product repository main branch, and serves for change propagation (for both parties). If DE changes something on it, this is an update propagation. Hence, update propagation looks like permitting DE to override assets in Product repositories, which seems risky. Scalability might be an issue as well.

Our work follows Anastasopoulos in so far as taping into existing VCS tools (in our case, *git* and *GitHub*). Like Thao et al., we also advocate for two types of repositories that are linked: *CoreAsset* repositories and *Product* repositories. Unlike Hellebrand et al. we advocate for a product repository that maintains the trace back to the *CoreAsset* repository it was derived from. *Product* repositories, will instantiate the core-asset release they were derived from, then, filter those not necessary for constructing the product, and only after making the pertinent product-specific changes, the product can be generated/built (preprocessed or composed). This approach would enable both update and feedback operations.

Figure 5.2 depicts our sample SPL arranged along this repository architecture. Each repository is a separated installation, hence, managed by its own team. However, the SPL's repositories are not isolated but conform an ecosystem tightened together through sync paths (depicted through dotted lines in Figure 5.2). Unfortunately, inter-repository operations are so far limited to *fork & pull model: a fork* clones a whole repository into a brand new one, which evolves independently until it might be merged back through *a pull.* This fits well for open source software projects but fall shorts for SPLs. Here, reuse is not based on whole cloning but derivation: cherry-picking core-asset and next, customization. On this premise, we introduce the *derive & update & feedback model* which rests in the namesake operations. Unlike *fork*, *derivation* does not involve a whole clone but a cherry-picking selection of core-assets. In the same vein, and in contrast with *GitHub's pull*, *update* & *feedback* govern a piecemeal synchronization between *Product* repositories and its source *CoreAsset* repository. Next section delves into the branching models that we propose.

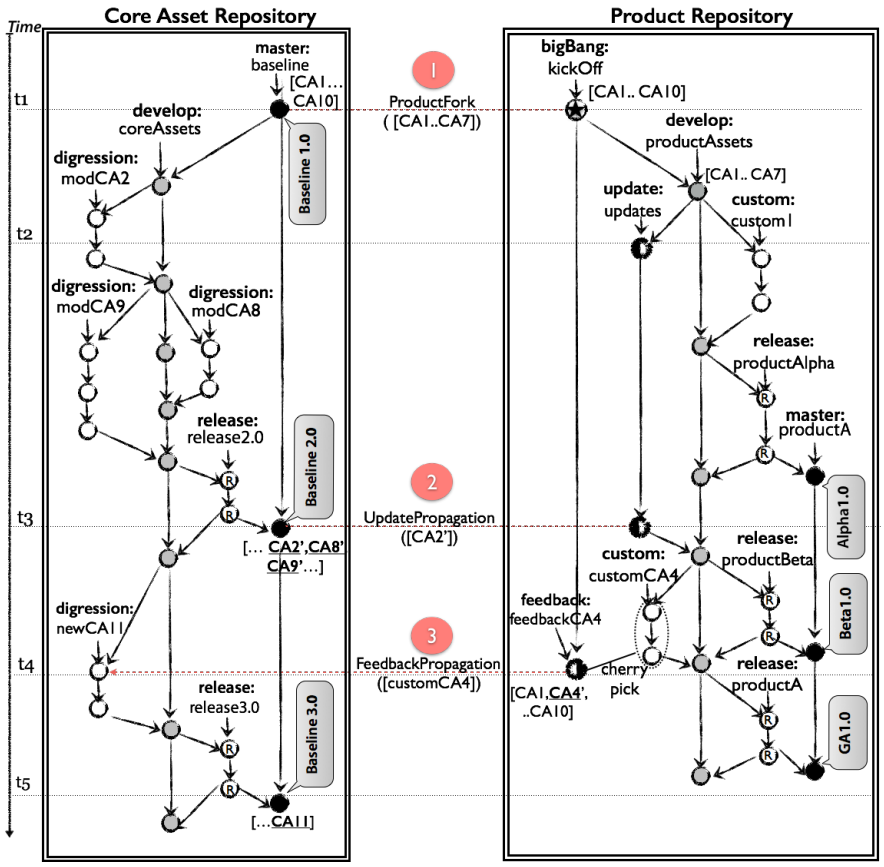

Figure 5.3: A closer look into the scenario described in Figure 5.2: branching impact due to (1) Product Fork, (2) Update Propagation and (3) Feedback Propagation. CA stands for the core-assets of the sample SPL.

5.5 Proposed branching models

VCSs support “revisions”, i.e. a line of development (the *baseline* or *trunk*) with branches off of this, forming a directed tree, visualized as one or more parallel lines of development (the "*mainlines*" of the branches) branching off a baseline (see Figure 5.3). The question is how to mimic the *modus operandi* of SPL development in terms of “parallel lines of development”, i.e. setting the branching model. Since core-assets and products are not born equal (i.e. products are derived from core-assets while core-assets might be obtained from scratch or extractively from existing products), we believe they can be better served by distinct branching models.

5.5.1 A Branching Model For Core-assets

For single-product development, a popular approach is *branch-per-purpose* [WS02a]. This strategy recommends different branch types per task type. A popular *git* branching model, *git-flow*, includes the following branching types [Gitb]: *master*, *develop*, *digression* and *release*. The usage of *git-flow* in industrial companies has been reported in [KDB+15a] . For understanding sake, we stick to this terminology (see Figure 5.3 top):

- *Master* branch is a long-lived branch aimed at core-assets release management. Each commit under *master*, holds a stable release of core-assets that work together (*e.g. Baseline 1.0* holds core-assets *CA1* to *CA10*). This branch, becomes essential for application engineers, and it is the cornerstone for product derivation[4].
- *Develop* branch is a long-lived branch which serves as the mainline of development for core-assets.
- *Digression* branches are short-lived branches that serve to assist on parallel development of core-assets, to create new core-assets or adapt existing ones (e.g., *updateCA2* branch enhances *CA2* core-asset).
- *Release* branches are short-lived branches used to prepare the next release for the core-asset baseline, before merging it to master (e.g., *release2.0* branch).

This approach accounts for a parallel and consistent development of core-assets under a single join development (by means of *Develop* and *Digression* branches). In addition, products can rely on a consistent release of core-assets (baseline release in *Master* branch). This model embraces a *release strategy* whereby all core-assets are made available *all* together on regular intervals. This may introduce a latency for application engineers. That is, even if a core-asset implementation is ready for production, it cannot be released until other core-assets are also ready to be in the next baseline release. This latency might lead product engineers to "clone and own" the best-fitting asset and adapt it to their needs [Mcg03]. Finding the right release pace is up to each SPL organization.

5.5.2 A Branching Model For Product Repositories

Unlike core-assets, products are derived from other artifacts, i.e. the core-assets. This states a dependency between products and core-assets. Better said, between a product and the core-assets used for its derivation. Notice, this dependency is not with *all* core-assets but just with those assets that participate in the initial product configuration. This dependency might involve for product engineers, first, to be aware of upgrades for the core-assets at hand *(update propagation)*, and second, being able to communicate product customization which might be amenable to be turned into SPL's core-asset *(feedback propagation)*. This subsection introduces a branching model conceived for facilitating these propagations. By "facilitating" we mean to be able to express those

[4]That the core-asset code is fully stable might be less an issue if development speed counts. Releasing not-fully tested features might make sense in these scenarios which we have not considered here.

propagations in terms of the basic VCS constructs (i.e. branch, merge, fork, pull). The final aim is to spot mismatches risen during synchronization *à la VCS*, i.e. highlighting *diff*-erence between distinct versions of the same artifact. In this way, SPL engineers handle sync in a very similar way to what they are used to for single products.

Our branching model for *Product* repositories rests on seven branch types to account for three purposes: development, delivery and propagation. For illustration purposes, we resort to our running example (see Figure 5.2) but now looking inside the repositories (see Figure 5.3).

For development: *BigBang, Develop & Custom* branches. *BigBang* is a long-lived branch, which keeps localized the *baseline* from which the product was derived. For instance, if a product wants to be derived from the CoreAsset *Baseline 1.0* , a *BigBang* branch would point to a commit exactly the same as *baseline 1.0* (same commit object, although in different repositories). This branch remains *untouched*, during the repository life time. This is so, to enable feedback propagation process (see later). On the other hand, *Develop* and *Custom* branches embrace parallel development for product assets. *Develop* branch is a long-lived branch which holds the mainline of product asset development. *Custom* branches, obtained off *Develop* branches, are used for product specifics: core-assets can be adapted while brand new assets can be introduced. When a customization is considered finished, *Custom* branches are merged back into *Develop* branch . Although good practices would advocate to delete *Custom* branches after merging them back to the mainline, our model maintains these branches alive for feedback purposes. Figure 5.3 (bottom) shows the case where a *Product* repository is derived from *Baseline1.0*, instantiating core-assets *CA1* to *CA7*. Additionally, *CA1* is customized to *CA1'*, hence giving rise to a *Custom* branch.

For delivery: *Release & Master* branches. Upon a consistent set of product assets under a *Develop* branch, *Release* branches are created for obtaining an executable product with the help of assembly tools. When this product is ready for GA Release, it would be merged to the *Master* branch and tagged accordingly. *Master* is a long-lived branch containing product releases ready to be delivered to customers. Figure 5.3, shows the case where *productA* alpha release consists of the initially derived core-assets plus *CA1'* customization. The beta release includes an additional enhancement on *CA2'*. Finally, the GA Release also comprises a customization for *CA4*'.

For propagation: *Update & FeedBack* branches. Parallel development involves resolving eventual conflicts when acting upon the same artifact. VCSs offer *diff* tools that highlight differences in code lines to easily spot mismatches. For these tools to be effective, the artifacts to be compared should correspond to versions of the same artifact. However, when an artifact is composed with other artifacts, the result can no longer be qualified as "a version" of the composing artifacts. Hence, applying *diff* between a core-asset and a product would be of limited use since the code of the core-asset might be tangled and polluted with code that is not related with the core-asset as such. This calls for *Product* repositories to keep an independent line of branching with *untouched* core-assets. This is the goal of *Update* branches: holding the product's core-assets separated from the product mainline (i.e. *develop* branch). Upon a new baseline release in the *CoreAsset* repository, product engineers might request an update propagation and easily spot differences using *diff* (see later).

Back to our example in Figure 5.3, domain engineers have been busy yielding

Baseline 2.0 where *CA2* is leveraged to automatically play a movie when the user selects it from a movie list (*CA8* and *CA9* have also been adapted). At time *t3*, application engineers conduct an *UpdatePropagation* upon *Baseline 2.0*. Should this upper version be integrated? The decision is twofold. First, product engineers *diff*-erentiate what's new w.r.t. to previous version (i.e. *diff(CA2, CA2')*). If satisfied, next they assess the impact of the new version of *CA2* with respect to the product as such. This implies a merge with a *Develop* branch (see Figure 5.3). This accounts for a *diff*-driven stepwise decision that might help spotting potential mismatches between how *CA2* evolve (in the domain realm) and how *CA2* was customized (in the product realm).

Finally, *Feedback* branches support promotion of meaningful product customizations into core-assets. By meaningful is meant a customization that makes sense as a unit. This might imply collecting code scattered throughout several *Custom* branches. The feedback process is twofold (see Figure 5.3). First, a *FeedBack branch* is created to *diff*-erentiate the customization code from the code in the original core-assets. To isolate the customization code (i.e. avoiding mixing it up with other functionality), we cherry-pick those changes from the *Custom* branch at hand [5]. Back to the example, *CA4* was customized to automatically re-play a movie after finished. At time *t4*, application engineers conduct feedback propagation. First, they need to pinpoint the *Custom* branches at hand (e.g., *customCA4* branch). Next, changes of *customCA4*, are cherry-picked and merged into a *FeedBack* branch. Hence, *feedbackCA4* branch only contains those changes for *customCA4* (i.e., *CA4'*). When domain engineers handle this feedback request, a *diff(develop:coreAssets, feedback:feedbackCA4)* will highlight only changes for the new functionality (i.e., *CA4*). Domain engineers can now decide to stick with *CA4* or rather, open a new core-asset (i.e., *CA11*) where to generalize the product customization to the whole SPL.

5.6 SPL sync operations as first-class constructs in VCSs

Previous section introduces branching models for *ProductFork*, *UpdatePropagation* and *FeedbackPropagation* to be expressed in terms of VCS primitive operations (i.e. fork, branch, merge). For instance, a *productFork* involves both a *fork* and a *branch*: a *fork* upon the CoreAsset repository which creates a *BigBang* branch; next, *BigBang* is branched into a *Develop* branch where only the required core artifacts are kept. Likewise, *UpdatePropagation* and *FeedbackPropagation* can also be expressed in terms of these VCS primitives. However, this introduces a gap between how operations are conceived, and how operations are realized, with the consequent costs associated. Our aim is to leverage existing VCSs with these operations as first-class constructs. To this end, we need first to precisely indicate their operational semantics, and next, to integrate them into a VCS tool. As a proof-of-concept, we outline a *GitHub*

[5] VCS's *cherry-pick* operation takes the changes introduced in a commit, and tries to reapply it on the *current* branch. This is useful when there is a number of commits on a branch, and only one of them is to be integrated into another branch.

Algorithm 5.1 Product Fork

```
1  ProductFork(UserAccount:userAccount,Repository:coreRepo,String
       []:configuration):Repository
2  Repository productRepo=Fork(userAccount,coreRepo)
3  productRepo.name=split(coreRepo.name,'-')[0]+'-Product-'+
       currentDate()
4  productRepo.description='A product derived from '+coreRepo.name
5  for each branch in productRepo.branches do
6    if (branch.name<>'master:baseline')
7              DeleteBranchByName(userAccount,productRepo,branch.
                   name)
8  Branch master=GetBranchByName(userAccount,productRepo,'master.
       baseline')
9  Branch bigBang=new Branch(userAccount, productRepo, master,'
       bigBang:kickOff')
10 Branch develop= new Branch(userAccount,productRepo,bigBang,'
       develop:productAssets')
11 SetDefaultBranch(userAccount,productRepo, develop)
12 DeleteBranchByName(userAccount,productRepo,'master:baseline')
13 Folder CRepBaseline=develop.commit.folders
14 for each coreAsset in CRepBaseline  do
15   if (coreAsset.name not in configuration)
16     DeleteFolder(userAccount,productRepo,develop,coreAsset)
17 Branch update=new Branch(userAccount,productRepo, develop,'
       update:updates')
18 File productConfig=new File(userAccount, productRepo,'product.
       config',bigBang.commit.sha)
19 Commit(userAccount,productRepo,update,productConfig,'Create
       config file')
```

implementation.

5.6.1 Product Fork

ProductFork takes a *CoreRepository* as input, and delivers a *ProductRepository*, along a given configuration. Namely:

> PRODUCTFORK (USERACCOUNT:USERACCOUNT, REPOSITORY:COREREPO, STRING[]: CONFIGURATION):: REPOSITORY:PRODUCTREPO

where USERACCOUNT stands for the application engineer's *GitHub* user account; COREREPO stands for the *CoreAssetRepository* from which a *Product* repository will be derived; and CONFIGURATION holds a list of core-asset identifiers. PRODUCTREPO stands for the newly initialized *Product* repository. Figure 5.4 describes the new *Product* repository. Algorithm 5.1 provides the details:

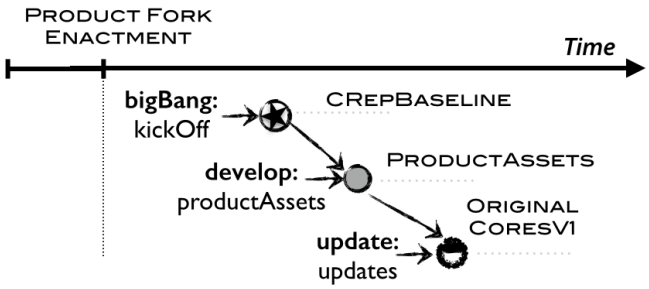

Figure 5.4: Product Fork involves 3 branches & 3 commits.

1. Perform a FORK operation over COREREPO (line 2). Now, USERACCOUNT owns a copy of COREREPO repository. At this point, PRODUCTREPO and COREREPO are identical (same branches, commits, tags, repository details, etc), except for PRODUCTREPO holds a *fork link* to COREREPO.

2. Rename PRODUCTREPO with pattern <*SPL_name*> <*product*><*date*>, and change its *description* to state that PRODUCTREPO is actually a product derived from a core repository (lines 3-4).

3. Adapt PRODUCTREPO to the product branching model introduced in section 5.5.2 (lines 5-19), namely:

   (a) First, all branches that PRODUCTREPO holds, are deleted (lines 5-7), except for *master : baseline* branch, which in ProductRepository turns into *bigBang: kickOff*. As there is no way to rename a branch in *git*, the way to simulate this operation is to, first create a new branch for *bigBang:kickOff* (lines 8-9), and then delete *master:baseline* (line12). *BigBang:kickOff* keeps now all core-assets from COREREPO baseline (i.e, CREPBASELINE). DELETEBRANCHBYNAME operation performs an HTTP request to delete branches of *GitHub* repositories.

   (b) Second, *develop:productAssets* branch is created off *bigBang:kickOff* (line 10). GETBRANCHBYNAME operation is accessed the *GitHub* API to obtain a branch by its name from a given repository. SETDEFAULTBRANCH operation performs a HTTP request to set as default branch of a *GitHub* repository.

   (c) Third, those core-assets not referred in CONFIGURATION are deleted (lines 13-16). DELETEFOLDER operation performs HTTP requests to delete all files from a given folder. At this point *develop: productAssets* branch only holds the core-assets needed to exhibit by the product (Figure 5.4, PRODUCTASSETS).

   (d) Finally, *update:updates* branch is created off *develop:productAssets* (line 17), and initialized with the *Product.config* file. This file holds

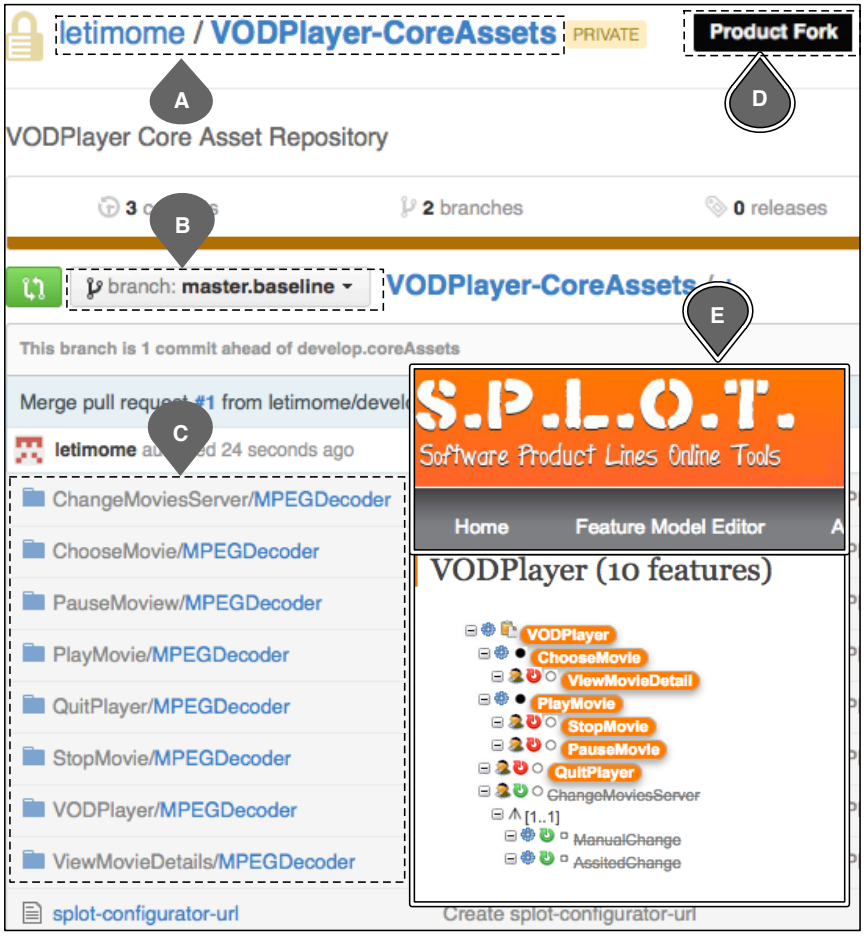

Figure 5.5: Leveraging *GitHub* with *ProductFork*

the *sha*[6] identifier of the CoreRepo's baseline version from which productRepo is derived (line 18-19). At this point, *update:updates* holds *original* reusable core-assets versions (Figure 5.4, OriginalCoresv1).

5.6.1.1 Leveraging *GitHub* with *ProductFork*

Product derivation is performed upon *CoreAsset* repositories. Figure 5.5 depicts *VODPlayer-CoreAssets* repository, which is available at the following link `https://github.com/letimome/VODPlayer-CoreAssets` . However, this will

[6]"sha" is *GitHub* name for unique *hash* identifier for an artifact, let this be a folder, a file or a commit object.

only recover a plain *GitHub* HTML page. Enhancing *GitHub* pages with SPL-specific VCS operations is achieved through the ***GitLine*** browser extension. ***GitLine*** makes on-the-fly changes to *GitHub* pages to account for *ProductFork*, *UpdatePropagation* and *FeedBackPropagation*. Using Web Augmentation techniques [DA15], *GitLine* adds buttons to enact those operations, i.e. repositories are accessed through *GitHub's* APIs, and extra *iFrames* are popped-up, should additional interactions with the user be needed. *GitLine* has been proven for Firefox 37.0, and its available for download at `http://onekin.github.io/GitLine/`. Note that *GitLine* needs to be locally installed in each browser from where the SPL repository is to be accessed. This subsection focuses on *ProductFork*. Drop-like icons are used to highlight certain facts. Double-lined drops denote *GitLine* layered content.

Figure 5.5 depicts *VODPlayer-PL CoreAsset* repository . Drop **A** points to the owner and repository name: *letimome* and *VODPlayer-CoreAssets*, respectively. Drop **B** points to the current branch. Drop **C** points to the core-assets. On top of this rendering, *GitLine* layers additional content: a new button (drop **D**). On clicking, a panel shows up which delivers an IFrame which holds the result of invoking a web-accessible feature configurator: *S.P.L.O.T* [S.P] (drop **E**). The panel is automatically generated from the *VODPlayer* feature model which, in the current implementation, needs to be previously loaded at *S.P.L.O.T.* Users are now guided by *S.P.L.O.T* in setting the configuration (in the screenshot core-assets *CA1* to *CA7* are selected). Once the configuration is over, the *ProductFork* algorithm resorts to *GitHub*'s APIs to automatically create a *GitHub* repository. Its name follows the pattern: <*SPL_name*><*product*><*date*> (e.g. *VODPlayer-Product-05ABR2015*). This repository is already initialized with a *BigBang* branch, *Update* branch and a *Develop* branch (Figure 5.4). The latter holds the selected core-assets. Now, application engineers are ready to start.

5.6.2 Update Propagation

UpdatePropagation takes a *Product* repository as input, and creates a new version for the *Update* branch. Namely:

> UPDATEPROPAGATION(USERACCOUNT: USERACCOUNT, REPOSITORY: PRODUCTREPO, REPOSITORY: COREREPO) :: PULLREQUEST

where USERACCOUNT stands for the application engineer's *GitHub* user account; PRODUCTREPO denotes the hosting ProductRepository; and COREREPO corresponds to the *CoreAsset* repository from which PRODUCTREPO was derived. The precondition to trigger the operation is: there is a new baseline version in COREREPO whose changes have not been yet propagated to PRODUCTREPO. This is assessed by reading PRODUCTREPO'S *product.config* file under *update:updates* branch, which holds the *sha* identifier to the COREREPO baseline to which PRODUCTREPO is currently synchronized. If the *sha* at *product.config* differs from the one at COREREPO'S *master:baseline*, it means that PRODUCTREPO is unsynchronized with COREREPO, and thus, update propagation can be enacted. Figure 5.6 describes *Product* repository

Algorithm 5.2 Update propagation algorithm.

```
1  UpdatePropagation(UserAccount:userAccount,Repository:
       productRepo,Repository:coreRepo):PullRequest
2  Branch update=GetBranchByName(userAccount,productRepo,'update:
       updates')
3  Branch coreBaseline=GetBranchByName(userAccount,coreRepo,'
       master:baseline')
4  Folder originalCoresV1=update.commit.folders
5  Folder originalCoresV2= null
6  for each coreAsset in originalCoresV1 do{
7    originalCoresV2= GetFolderByName(userAccount,coreRepo,
         coreBaseline,coreAsset.name)
8    if (coreAsset.sha<>originalCoresV2.sha)
9      CommitFolder(userAccount,productRepo,update,originalCoresV2
           ,'new update for core asset:'+originalCoresV2.name)
10 }
11 File productConfig=GetFileByName(userAccount,productRepo,update
       ,"product.config")
12 productConfig.content=coreBaseline.commit.sha
13 Commit(userAccount,productRepo,update,productConfig,'product
       synched to baseline'+coreBaseline.commit.sha)
14 Branch develop=GetBranchByName(userAccount,productRepo,'develop
       :productAssets')
15 CreatePullRequest(userAccount,productRepo,productRepo,develop,
       update,update.commit.comment)
```

branching structure before and after the operation. Algorithm 5.2 describes the operational semantics:

1. Get the latest baseline version available from CORREPO, and bring to PRODUCTREPO'S *update* branch the newest versions of those core-assets that PRODUCTREPO is reusing (lines 2-10).

   (a) Specifically, for all those core-assets versions PRODUCTREPO is currently reusing (i.e, ORIGINALCORES -V1), check if there is a newer reusable core-asset version at COREREPO (lines 6-8).

   (b) If there is a newer version, get it and *commit* the new version of the asset (i.e., ORIGINALCORES-V2) to *update:updates* branch (line 9). As *GitHub* web site only allows to commit a single file at a time, we developed COMMITFOLDER HTTP operation which given a folder, all files contained inside are committed iteratively. At this point, PRODUCTREPO holds new versions of reusable core-assets under *update:updates* branch (i.e., ORIGINALCORESV2).

2. Update *product.config* file to indicate that PRODUCTREPO is now in sync with PRODUCTREPO (lines 11-13). First the file is obtained by means of

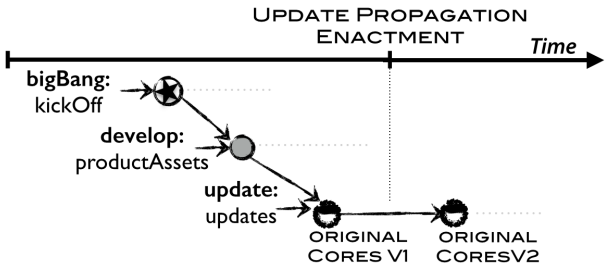

Figure 5.6: Update Propagation involves 1 commit for each core-asset updated core-asset & 1 pull_request

GETFILEBYNAME operation, which is a HTTP request to get a file from a *GitHub* repository (line 11). Afterwards, file content is updated with the *sha* identifier of COREREPO last baseline version (line 12), and committed to *update:updates* branch (line 13).

3. Finally, a *pull request* is enacted to notify application engineers about the new changes pulled from the *CoreAsset* repository (lines 14-15). The *pull request* requests to merge *update: updates* branch into *develop: productAssets* branch. At this point application engineers can reason about the impact of this updates have into the product assets by popping up the *diff* panel (see later).

5.6.2.1 Leveraging *GitHub* with *UpdatePropagation*

Update propagation is performed by application engineers upon a *Product* repository. Figure 5.7 depicts *VODPlayer-Product-05ABR2015,* i.e. the *Product* repository obtained in the previous sub-section, available at `https://github.com/lemome88/VODPlayer-Product-05ABR2015`. Let's assume that core-assets evolve until *Baseline 2.0* (time frame t1-t3) where a new version of *CA2* (i.e. *ChooseMovie*) is available. During the same timeframe, product engineers customized *CA1* into *CA1'*. At this time, application engineers perform *updatePropagation*. Figure 5.7(left) depicts this scenario. Drop **B** points to the current branch. Drop **A** points to the *Update_Propagation* button. On clicking, a pop-up displays the summary of changes to be pulled (drop **C**): a list of rows with the name of the updated core-asset (e.g. *ChooseMovie*), and a *link* to the Core-Asset-repository's commits describing those changes (*"New commits"*). Following these links brings product engineers to the Core Asset realm by opening a new browser tab, where the *ChooseMovie* asset evolution is shown in a diff panel (not shown in the Figure), so that product engineers can make an informed decision about whether to pull these changes back to the *Product* repository. If so decided, developers go back to the *Product* repository (Figure 5.7(left), and click the *Yes* button (drop **D**). The *ChooseMovie* newer version is pushed to the *Update branch (e.g. update:updates*). Application engineers are notified through a new pull request (drop **E**) to merge *update:updates* into *develop:productAssets*. Developers can

Algorithm 5.3 Feedback propagation algorithm.

```
1  FeedbackPropagation(UserAccount:userAccount,Repository:coreRepo
       ,Repository:productRepo,Branch[]: customizations,String:
       feedbackBranchName)
2  Branch bigBang=GetBranchByName(userAccount,productRepo,"bigBang
       :kickOff")
3  Branch newFeedback=new Branch(userAccount,productRepo,bigBang,
       feedbackBranchName)
4  for each custom in customizations do {
5    Folder customizedAssets=GetChangesFromBranch(userAccount,
         productRepo,custom)
6    for each  custAsset in customizedAssets do
7      CommitFolder(userAccount,productRepo,newFeedback,custAsset,
           'customized asset:'+custAsset.name)
8  }
9  Branch develop=GetBranchByName(userAccount,coreRepo,'develop:
       coreAssets')
10 CreatePullRequest(userAccount,coreRepo,productRepo,develop,
       newFeedback,newFeedback.commit.comment)
```

now open the pull request to retrieve the changes (drop **F**). A new page shows up with the *diff*-erences: *diff*(*develop: productAssets*, *update*:*updates*). If changes are accepted, application engineers merge the branches. Otherwise, the pull request is closed, and the *Product* repository sticks with the *old* asset versions.

5.6.3 Feedback Propagation

FeedBackPropagation takes a *Product* repository as input, and creates a new version for the *FeedBack* branch. Namely:

> FEEDBACKPROPAGATION(USERACCOUNT: USERACCOUNT, REPOSITORY: COREREPO, REPOSITORY: PRODUCTREPO, BRANCH: KICKOFF, BRANCH[]: CUSTOMIZATIONS, STRING: FEEDBACKBRANCHNAME):: PULLREQUEST

where USERACCOUNT stands for the application engineer's *GitHub* user account; COREREPO stands for the *CoreAsset* repository; CUSTOMIZATIONS correspond to the set of branches that keep the product specific changes that want to be propagated back to the *CoreAsset* repository; finally, FEEDBACKBRANCHNAME refers to the name for the *feedback* branch to be created. Figure 5.8 describes *Product* repository branching structure before and after the operation. Algorithm 5.3 provides the details:

1. Create a *FeedBack* branch , labeled FEEDBACKBRANCHNAME (i.e., NEWFEEDBACK), off *bigBang:kickOff* (lines 2-3).

2. Build the meaningful customization based on existing CUSTOMIZATIONS branches (lines 4-8). This requires, for each *custom* branch in CUSTOMIZATIONS

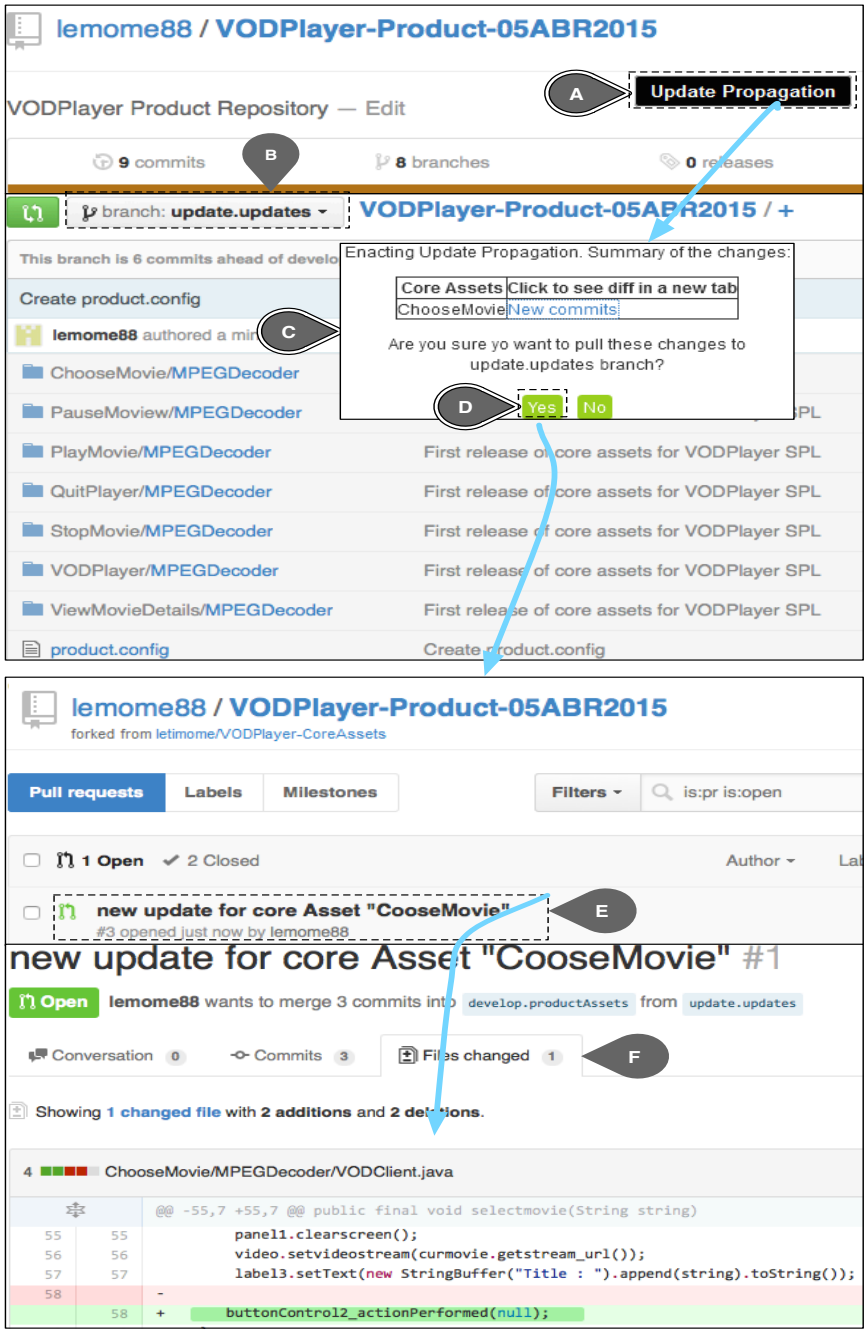

Figure 5.7: Leveraging *GitHub* with *UpdatePropagation:* enacting (top) and outcome (bottom).

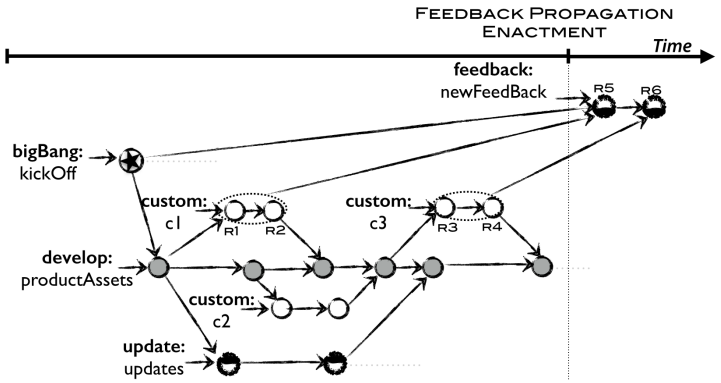

Figure 5.8: Feedback Propagation involves 1 branch & 1 commit for each Custom branch involved & 1 pull_request

(Figure 5.8, C1 AND C3), to *cherry-pick* the changes that each of the custom branch introduces (i.e., R1,R2 for C1) and to commit them into NEWFEEDBACK branch (i.e., R5). As *GitHub* does not provide cherry picking operation, we needed to develop it for *GitHub* repositories.

(a) This, requires first to identify the assets that a given branch (e.g, *custom* branch) has changed. GETCHANGESFROMBRANCH is a HTTP operation which returns all the artifacts that a given branch has changed, arranged in a tree structure.

(b) Then, all the identified assets are committed into NEWFEEDBACK branch.

3. When all CUSTOMIZATIONS have been merged into NEW FEEDBACK branch, a pull request is created in COREREPO, requesting to merge PRODUCTREPO'S NEWFEEDBACK branch into COREREPO *develop* branch (lines 9-10).

5.6.3.1 Leveraging *GitHub* with *FeedBackPropagation*

FeedBack propagation is performed over a *Product* repository. Figure 5.9 (left) depicts *VODPlayer-Product-05ABR2015* repository at time *t3*: a custom branch (i.e., *customCA4*) was created for *CA4* (i.e. *PlayMovie*). Meanwhile, *VODPlayer-CoreAsset* repository also committed some changes. At this point, application engineers want to promote changes done in *customCA4* (i.e. new version for *CA4*). Figure 5.9(left) depicts this scenario. Drop **B** points to the current branch. Drop **A** points to the new *FeedBack_Propagation* button. On clicking, a pop-up lists all *Custom* branches that the *Product* repository holds (drop **C**). Users can now select the desired customization (e.g. *customCA4* branch), and press the *Yes* button (drop **D**). This triggers the *feedback propagation* algorithm. Behind the scenes, a new *FeedBack branch* is created (i.e, *feedbackCA4*), and the *CoreAsset* repository receives a pull request coming from the *Product* repository (drop **E** in Figure 5.9(right)). When this request is opened, domain

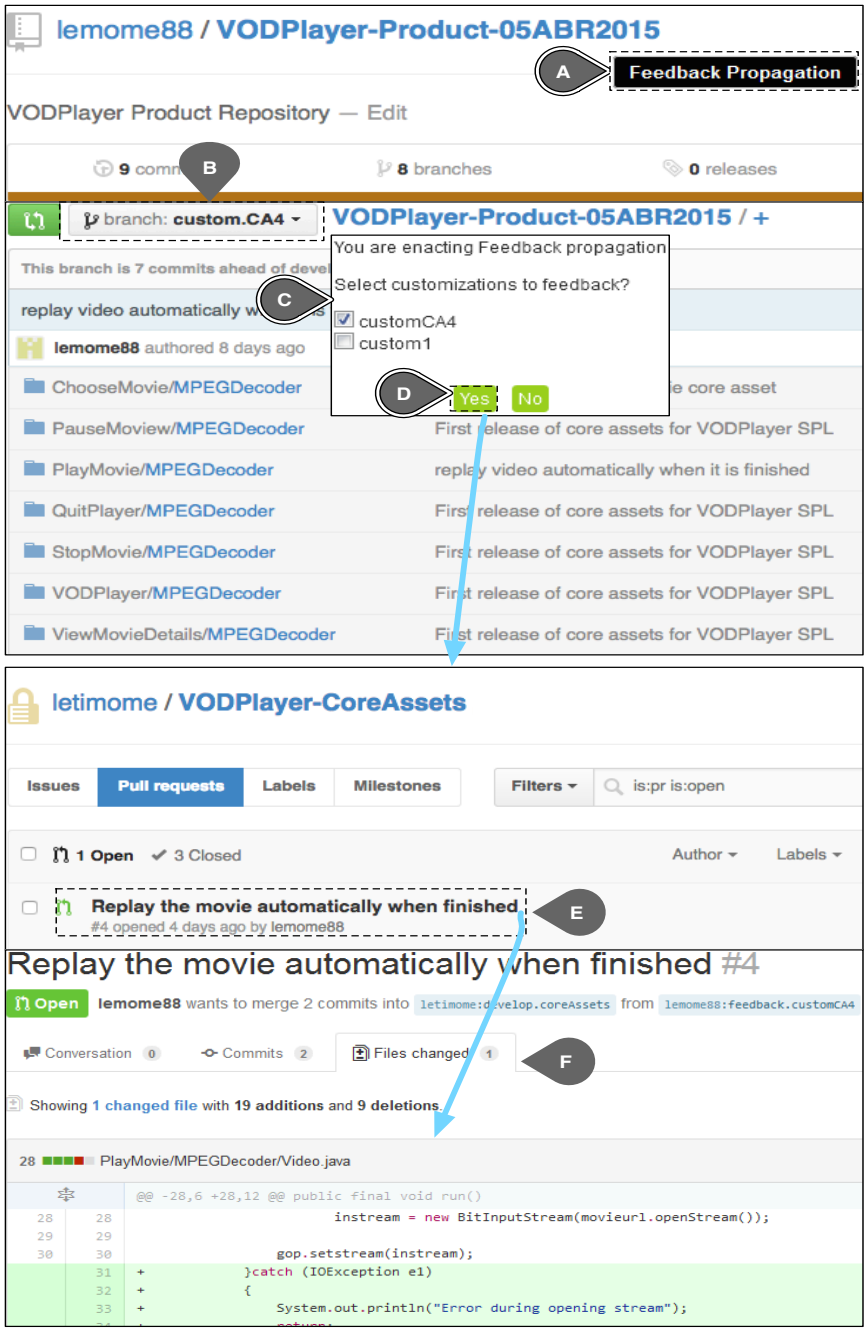

Figure 5.9: Leveraging *GitHub* with *FeedBackPropagation:* enacting (top) and outcome (bottom).

engineers are invited to merge the newly created *VODPlayer-Product's FeedBack branch* (i.e., *feedback:customCA4*) into *VODPlayer-CoreAssets' Develop* branch (i.e., *develop: coreAssets*). Drop **F** points to the diff *view* of the changes proposed by this pull request. At this point, domain engineers should decide whether the customization is useful to the whole product line. If so, domain engineers would need to refactor the customized core-assets. This might require to create a new *Digression* branch (e.g. *newCA11* branch in Figure 5.3).

5.7 Conclusion

This Chapter considers a SPL scenario where core-assets and products evolve along different life-cycles but get synchronized through propagation events. However, synchronization is achieved not between artifacts but artifact versions. This requires propagations to act upon the right version of artifacts. Specially, 3-way merges are required in order to not override changes done by the other party. We introduced a branching model that permits to capture sync paths in terms of VCS standard operations. Next, so-described processes are delivered as first-class constructs on top of an existing VCS, i.e. *Git*/*GitHub*. This permits reducing "the accidental complexity" that goes with supporting sync paths while freeing up developers for focusing on "the essential complexity", i.e. attuning and refactoring code coming from different developers. Tested for a FOP composer, the approach is valid as long as dedicated core-assets for dedicated functionalities are involved. Usability wise, the enhanced *GitHub* (i.e. the one augmented with *GitLine*) certainly outperforms the raw *GitHub*, if only as for reducing the number of clicks. *GitLine* is being used for two SPLs. Our hope is that by delivering *GitLine* to the community, sync-path good practices emerge. This work is an attempt to make these practices explicit and available. Future work includes to extend *GitLine* with composition options, facing scalability issues (i.e. SPL with large number of features and products), and evaluating *GitLine's* branching models and sync operations in industry. This would require to find a company where development efforts are carried out in both DE and AE. Our intuition is that SPLs at different levels of reuse might very well require different branching models. In a similar vein, different sizes of SPLs, as well as, differences in the business organization demand differences on the repository structures. For instance, for a big SPL it might be convenient to have a repository per reusable component. Nevertheless, in such a case, branching models are also necessary. But which one?. To the best of our knowledge no discussion is given on this issue. We aim at investigating this concern further.

Chapter 6

Conclusions

6.1 Overview

Following the so called *grow-and-prune* model [FV03] SPLs can be incrementally evolved by letting products *grow* and later *prune* product functionalities deemed useful by refactoring and merging. In this context, this Thesis investigates how current VCSs can be leveraged to support these practices. Specifically, we focus on the "prune" stage where Domain Engineers need to recap and merge what Application Engineers have been doing during the "grow" state. This includes capabilities for customization analysis, code peering and change propagation. The rest of the chapter reviews the main results of the Thesis, lists its limitations, as well as, new areas for future research are suggested.

6.2 Results

The contribution of this Thesis has been presented in the four central chapters of this manuscript. Next, we provide a summary for each:

- Chapter 2 revisits the concept of "evolution" in SPLs. This chapter systematically maps the existing research on SPL evolution, along four main facets. Well-covered areas, as well as, areas that require further research are identified. Analyses of the results indicate that "Solution proposals" are the most common type of contribution (31 %). However, few studies do address solutions for co-evolving core-assets and products. This fact, together with the evidences coming from the industry that attest the need for co-evolving core-assets and products (specially during the first years of the SPL life-cycle), grounds the importance of tackling co-evolution issues in SPLs. Specifically, few efforts have been made in order to identify product customization, as well as, to synchronize core-assets and products. This Thesis faces these gaps.
- Chapter 3 tackles customization analysis. We propose a data-warehouse approach to track product customization efforts. More concretely, we conduct a

survey among Danfoss drives engineers in order to identify the information needs required for conducting customization analysis. Next, we resort to Dimensional Modeling to tackle these information needs using the modified LOCs as *facts*. Finally, we propose the use of Alluvial diagrams as a visualization mean. This approach is fleshed out in *CustomDIFF*, a data-warehouse tool that uses *Git* as the operational system, and *pure::variants* as the SPL framework. Primary evaluations reveal promising results on *CustomDIFF*'s usefulness for customization analysis.

- Chapter 4 tackles the merge problem that arises during the pruning (i.e. merging and refactoring) of product customizations. We propose a new practice, i.e. code peering practice, as a way to lessen the issue by promoting early reuse across product teams right at product development. We discuss four design principles that drive how code peering can be introduced for SPL development. As a proof-of-concept we developed *PeeringHub*, a tool tool that supports code peering through: (1) enhancing Github with a peering bar, (2) exercising a DW solution similar to CustomDIFF's, and (3) leveraging feature-based 3-way comparison&merging. Primary evaluations reveal promising results with respect to usability and ease of use.

- Chapter 5 addresses *update* and *feedback* propagations. Branching models for SPL development are proposed, that permit to capture the sync paths in terms of Version Control System (VCS) standard operations. On these grounds, sync operations are delivered as first-class constructs. The approach is fleshed out for *GitHub*. This permits reducing "the accidental complexity" that goes with supporting sync paths while freeing up developers for focusing on "the essential complexity", i.e. attuning and refactoring code coming from different developers. Tested for a FOP composer, the approach is valid as long as dedicated core assets for dedicated functionalities are involved.

6.3 Publications

Part of the work presented in this thesis has been already presented and discussed in distinct peer-reviewed forums. The publications that endorse this Thesis are listed below.

Selected publications

- Leticia Montalvillo, Oscar Díaz: *Requirement-driven evolution in software product lines: A systematic mapping study*. Journal of Systems and Software (JSS). Volume 122, pages 110-143 (2016). DOI https://doi.org/10.1016/j.jss.2016.08.053. Related to Chapter 2.

- Leticia Montalvillo, Oscar Díaz, Maider Azanza: *Visualizing product customization efforts for spotting SPL reuse opportunities*. In the proceeding of the International Workshop on Reverse Variability Engineering (REVE'17),

full paper. Pages 73-80 (2017). DOI https://doi.org/10.1145/3109729.3109737. Related to Chapter 3.

- Leticia Montalvillo, Oscar Díaz, Maider Azanza: *CustomDIFF: A tool for customization analysis in SPLs.* In the proceedings of the International Conference on Software Product Lines (SPLC'18), tool paper. Related to Chapter 3.
- Leticia Montalvillo, Oscar Díaz and Thomas Fogdal: *Reducing Coordination Overhead in SPLs: Peering in on Peers*. In the proceedings of the International Conference on Software Product Lines (SPLC'18), full paper. Related to Chapter 4.
- Leticia Montalvillo, Oscar Díaz: *Tuning GitHub for SPL development: branching models & repository operations for product engineers*. In the proceeding of the International Conference on Software Product Lines (SPLC'15), full paper. Pages 111-120 (2015). DOI https://doi.org/10.1145/2791060.2791083. Related to Chapter 5.

Publications under review

- Oscar Díaz, Leticia Montalvillo, Maider Azanza: *The role of customization analysis for Software Product Line evolution*. Sent to the Special Issue on Software Product Line Engineering, in the Journal of Systems and Software (JSS) on January 2018. Related to Chapter 3.

6.4 Research visits

During the development of the Thesis I often found myself wondering about how a company that develops software with an SPL approach does "this" or "that". Experience reports that industries publish might shed some light... sometimes, but they frequently did not provide me with the answers I was looking for. Most of the time I ended up with even more questions... Therefore, I really wanted to meet *real* practitioners, from *real* companies, developing *real* SPLs. The chance turned my prays into reality. During the Software Product Line Conference (SPLC) held in Nashville in 2015 I met Thomas Fogdal, a functional manager (a.k.a. my golden pass to a SPL-developing company) working for Danfoss Drives. We soon found common interests and thanks to that I had the pleasure to perform a research visit at Danfoss Drives for 4 months. I had the opportunity to help them during the first steps of transitioning their SPL from their old ClearCase set-up, to a git-based set-up. During this period of time I had the chance to learn how a *real* SPL looks like, and how *real* SPL engineers work together to deliver software following an SPL approach. This insights were priceless, and certainly gave my Thesis a ready start. After my stay, I was able to come back again to Danfoss, in order to test out one of our new ideas (*i.e. CustomDIFF*) in their setting.

6.5 Assessment and future research

The goal of a Thesis is to try to resolve a problem. Nevertheless, in its development, some issues might remain open. We next assess the limitations of each piece of work presented in this Thesis and, we expose some of the topics that this Thesis leaves open. Discussion is articulated as per piece of work. Bullets list the limitations/future research opportunities.

Mapping the existing literature on SPL evolution

- Performing in-deep Systematic Literature Reviews (SLRs). SLRs are a form of more focused literature reviews, with a narrower scope and more specific research questions compared to SMSs. Our SMS was broad, as SMS are in nature. We provide an overview of the existing literature on SPL evolution, and we classified it along four main evolution activities, i.e. identify, analyze&plan, implement and verify change. Researchers can tap into our SMSs to conduct more in-deep SLRs. Potentially, a SLR for each evolution activity could be conducted.

Aiding SPL engineers conduct customization analysis

- Evaluating *CustomDIFF* in different SPL set-ups. The evaluation we have carried out at Danfoss Drives permitted us to assess the usefulness of *CustomDIFF's* with respect to analyzing product customization. However, we would like to further evaluate *CustomDIFF* in different companies to measure its effectiveness along two parameters: the SPL *maturity* (less mature SPLs might face higher customization effort) and the SPL *size* (the larger the number of core asset and products, the more compelling the need for abstract visualizations).

- Performing an experiment to evaluate *CustomDIFF's* usefulness to plan the next SPL release. The evaluation we have carried out at Danfoss Drives permitted us to assess the usefulness of CustomDIFF's to analyze product customization. However, we did not evaluate whether CustomDIFF help engineers plan the next SPL release. We would like to deploy it and evaluate its effectiveness to help engineers plan the next SPL release.

- Integrating *CustomDIFF* with other sources of information. We have so far focused on two dimensions of customization analysis: "the what" (i.e. what features & core assets are customized) and "the where" (i.e. products that performed such customization). It would be of interest to study how to supplement *Git* data with data coming from other sources, to collect information about products, customers and developers, and to see what other kind of analyses this additional sources would allow for. After all, data-warehouses are thought for integrating heterogenous data sources.

- Integrating *CustomDIFF* into a DevOps framework. *CustomDIFF* is an analysis tool and hence, it does not preclude the customization practice as such, in the

sense of determining how to proceed during the pruning phase. An interesting development would be using *CustomDIFF* within a DevOps framework where the customization effort (at its different abstraction levels) is tracked, and reactions can be attached to a certain customization-effort threshold being surpassed. Other scenarios include the use of *CustomDIFF* by product engineers to gaze what other mates are customizing. For instance, a feature enhancement introduced in a given product might be promptly and directly incorporated into other products, without waiting for this enhancement to be promoted as a core asset. This opens new scenarios for SPL evolution where "longitudinal evolution" (between core assets and products) might well co-exist with "traversal evolution" where products sharing same features might decide to incorporate enhancements from other products, and later on, be jointly pruned.

Fostering product engineers on peering into other peers

- Performing an experiment to evaluate *PeeringHub's* effectiveness to alleviate the merge problem during the pruning of product customization. The evaluation we carried out permitted us to assess the usefulness of *PeeringHub* to compare product customizations between them. However, we did not evaluate it at an industrial setting, nor we did evaluate whether code peering with *PeeringHub* lessens the merge problem.

Supporting the synchronization of core-assets and products

- Evaluating *GitLine's* branching models and sync operations. We have not yet evaluated the suitability of the proposed branching models and operations for synching core-assets and product repositories. The proposed branching models are suited for SPLs at a maturity level in which both core-assets and products require development. We did not have the chance to meet a company in such SPL maturity level willing to evaluate our approach.

- Providing support for other variability realizations. The presented approach was tested for a FOP composer (i.e. FeatureHouse). Nevertheless, the approach is valid as long as dedicated core assets for dedicated functionalities are involved, i.e. for composition-based approaches. The reader might have noticed how in this Thesis different variability implementation techniques were considered. Initially, we started this Thesis with the present piece of work, i.e. supporting the synchronization of core-assets and products (Chapter 4). Note, that the order in which the Thesis' chapters are arranged differ from the chronological order in which they were worked. At the moment we tackled this piece of work, evidences in the SPL literature showed how composition-based approaches outperformed annotation-based ones (at least for evolution related tasks). Hence, we opted to ground the work in a composition-based setting. Later in time, we conducted a SMS on SPL evolution, and this was the inflection point. The results of the mapping showed that most of the SPL-developing companies are not

using composition-based approaches, but annotation-based ones. Additionally, only few pieces of work addressing composition-based approaches were actually evaluated into an industrial context. That is the reason why our next research efforts (presented in Chapter 3) focuses on annotation-based approaches. With this paradigm switch we aimed to cause a significant impact on both the SPL research and practice.

- Comparing the suitability of different branching models for SPL development. The reader might have noticed, how in Chapter 3 and Chapter 4 we considered different repository structures and branching models for SPL development. While in Chapter 4 we advocated for a *separated repository* approach where core-assets and products are developed in separate repositories, in Chapter 3 we advocated for a *single repository* approach where both core-assets and product are being developed in the same repository but in different branches. This switch was due to the fact that the piece of work in Chapter 3 was motivated by the Danfoss experience. Since at Danfoss development in AE is minor (a two week development period), and no clear separation between DE and AE exists, a single repository approach suits well. However, this fact does not invalidate the repository model proposed in Chapter 4, since this can suit SPL companies with a clear separation between DE & AE, and which have a higher volume of development in AE. In fact, our intuition is that SPLs at different levels of reuse might very well require different branching models. In a similar vein, our intuition is that differences on the SPL size, as well as, differences in the team's organizational model of an SPL, demand different VCS repository structures. For instance, for a big SPL in which teams are organized around the SPL high-level components, it might be convenient to have a repository per reusable component. No matter the repository structures, underlaying branching models that drive development are also necessary. But which pair of branching model and repository structure best fits the development of an SPLs? To the best of our knowledge no discussion is given on this issue. We aim at investigating this concern further.

6.6 Conclusion

This Thesis took a Design Science Research approach to identifying and solving problems that raise when incrementally evolving SPLs from product developments. In order to evaluate the state of-the-art on the area of SPL evolution, we systematically mapped the existing literature on the topic. This, helped us to spot the fact that few efforts were made to address the co-evolution of core-assets and products. This finding kicked-off this Thesis, and provided three main issues to investigate: (1) how to help SPL engineers analyze product customizations, (2) how to alleviate the merge problem that raises during the pruning of product customization, and (3) how to help SPL engineers synchronize core-assets and products. Following good practices for design science research,

- the importance of the problems were argued,

- root-cause analysis was conducted for these problems,
- three artefacts (i.e. *CustomDIFF*, *PeeringHub*, and *GitLine*) were developed to lessen some of these causes (i.e. lack of dedicated visualization tools for customization analysis, low abstraction level at which customization analysis is conducted, large amount of conflicts between product customizations, VCSs not tuned to SPL propagation operations, and lack of guidelines for branching and merging).
- those artefacts were evaluated to the extent research prototypes can be evaluated in an industrial SPL setting,

There is a considerable amount of future work. We invite both the product line research and the industry to join our efforts and further improve the work started in this Thesis, by refining the proposed approaches to support the *grow-and-prune* model, studying their applicability to different SPL contexts and scenarios, and providing missing solutions.

Although this chapter "physically" concludes this dissertation, the journey continues.

Appendix A

Papers on SPL evolution classified on facets

Ref.	Title	Year	Evolution activity	Evolution sub-activity	Asset type	Product-derivation appr.	Research type
[KSS15]	A process to support a systematic change impact analysis of variability and safety in automotive functions	2015	Analyze and plan	Change impact	Products	Model-driven	Solution
[HVLG12]	A case study on the evolution of a component-based product line	2012	Analyze and plan	Change impact	Variability model, SPL architecture, Code assets	Composition	Experience
[Sch06a]	A cost-based approach to software product line management	2006	Analyze and plan	Decision-making	Variability model	NA	Conceptual
[BM14]	A cover-based approach for configuration repair	2014	Implement	Change synchronization	Variability model	NA	Validation
[MKR94]	A holistic approach to product marketability measurements-the PMM approach	1994	Analyze and plan	Decision-making	Products	NA	Solution
[KB13]	A mixed-method approach for the empirical evaluation of the issue-based variability modeling	2013	Analyze and plan	Decision-making	Variability model	NA	Evaluation
[TABG15]	A product line of theories for reasoning about safe evolution of product lines	2015	Implement	Built-with-change	Variability model, SPL architecture,	Hybrid	Solution
			Verify	Inconsistency checking			

					Code assets		
[GFFd14]	A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines	2014	Implement	Built-for-change	Code assets	Composition	Evaluation
[Kla08]	A requirements-based taxonomy of software product line evolution	2007	Identify	Monitoring the environment	NA	NA	Conceptual
[BTG12]	A theory of software product line refinement	2012	Implement	Built-with-change	Variability model, Code assets	Hybrid	Solution
			Verify	Inconsistency checking			
[DLS05]	Addressing domain evolution challenges in software product lines	2006	Implement	Built-for-change	SPL architecture	Model-driven	Solution
[DPG14]	Agile product-line architecting in practice: A case study in smart grids	2014	Analyze and plan	Change impact	SPL architecture	Composition	Evaluation
			Implement	Built-for-change			
[NRG08]	Agile product line planning: A collaborative approach and a case study	2008	Analyze and plan	Planning	NA	NA	Validation
[CdOW11]	An analysis of change operations to achieve consistency in model-driven software product lines	2011	Implement	Change synchronization	Code assets	Model-driven	Conceptual
[SS08]	An analysis of effort variance in software maintenance projects	2008	Analyze and plan	Decision-making	Code assets, SPL architecture, Code asset	NA	Solution
[TM14]	An approach for decision support on the uncertainty in feature model evolution	2014	Analyze and plan	Decision-making	Variability model	NA	Solution
[GCC+03]	An environment for managing evolving product line architectures	2003	Implement	Built-with-change	SPL architecture	Composition	Solution
[ddC+12]	An experimental study to evaluate a SPL architecture regression testing approach	2012	Verify	Inconsistency checking	SPL architecture, Code assets	Composition	Evaluation
[CDG+12]	Analysing the impact of feature dependency implementation on product line stability: An exploratory study	2012	Implement	Built-for-change	Variability model, Code assets	NA	Evaluation

[PYZ11]	Analyzing evolution of variability in a software product line: From contexts and requirements to features	2011	Analyze and plan	Decision-making	Variability model	NA	Conceptual
[IKH14]	Application of requirements prioritization decision rules in software product line evolution	2014	Analyze and plan	Planning	NA	Composition	Experience
[VGH+12]	Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines	2012	Verify	Inconsistency checking	Variability model, Code assets	Model-driven	Evaluation
[DKO+97]	Applying software product-line architecture	1997	Implement	Built-for-change	SPL architecture	NA	Experience
[GF13]	Architectural evolution of FamiWare using cardinality-based feature models	2013	Analyze and plan	Decision-making	Variability model	NA	Validation
			Implement	Change synchronization			
[MV09]	Aspect-oriented change realization based on multi-paradigm design with feature modeling	2012	Implement	Built-with-change	Code assets	Composition	Solution
[tBMP12]	Assume-guarantee testing of evolving software product line architectures	2012	Verify	Scalable verification	SPL architecture	Composition	Solution
[DLHE14]	Automatic and incremental product optimization for software product lines	2014	Verify	Inconsistency checking	Variability model, Code assets	Composition	Validation
[RRSW]	Behavioral compatibility of simulink models for product line maintenance and evolution	2015	Verify	Scalable verification	SPL architecture	Composition	Validation
[KR13]	Bi-criteria genetic search for adding new features into an existing product line	2013	Analyze and plan	Decision-making	Variability model, Code assets	NA	Solution
[Hol12]	Challenges in the evolution of model-based software product lines in the automotive domain	2012	Implement	Change synchronization	SPL architecture	Composition	Experience
[PDŠ12]	Change impact analysis of feature models	2012	Analyze and plan	Change impact	Variability model	NA	Solution
[SHA12]	Co-evolution of models and feature mapping in software product lines	2012	Implement	Change synchronization	Variability model, Code assets	Model-driven	Solution

[PGT+13]	Coevolution of variability models and related artifacts: A case study from the Linux kernel	2013	Implement	Change synchronization	Variability model	Annotation	Validation
[CL01]	Comparing frameworks and layered refinement	2001	Implement	Built-for-change	Code assets	Hybrid	Evaluation
[AKEs12]	Comparing maintainability evolution of object-oriented and aspect-oriented software product lines	2012	Implement	Built-for-change	Code assets	Composition	Evaluation
[TDR+11]	Components meet aspects: Assessing design stability of a software product line	2011	Implement	Built-for-change	SPL architecture	Composition	Evaluation
[QPB+14]	Consistency checking for the evolution of cardinality-based feature models	2014	Verify	Inconsistency checking	Variability model	NA	Validation
[GWTB12]	Consistency maintenance for evolving feature models	2012	Implement	Change synchronization	Variability model	NA	Evaluation
[LLSG12]	Delta-oriented model-based SPL regression testing	2012	Verify	Scalable verification	Products	Compositon	Solution
[SBB+10]	Delta-oriented programming of software product lines	2010	Implement	Built-for-change	Code assets	Composition	Evaluation
[TBM+12]	Developing long-term stable product line architectures	2012	Implement	Built-for-change	SPL architecture	NA	Experience
[CCG+03]	Differencing and merging within an evolving product line architecture	2004	Implement	Change synchronization	Products	Composition	Solution
[DKvDP15]	Evaluating feature change impact on multi-product line configurations using partial information	2015	Analyze and plan	Change impact	Variability model	NA	Validation
[VDJ10]	Evaluation of a method for proactively managing the evolving scope of a software product line	2010	Identify	Monitoring customer	NA	NA	Evaluation
[TB07]	Evolution in product line requirements engineering: A rationale management approach	2007	Analyze and plan	Decision-making	NA	NA	Validation
[PHS11]	Evolution patterns for business document models	2011	Analyze and plan	Change impact	Variability model	Model-driven	Solution
[Tes07]	Evolving embedded product lines: Opportunities for aspects	2007	Implement	Built-for-change	Code assets	Composition	Experience
[FCS+08]	Evolving software product lines with aspects	2008	Implement	Built-for-change	Code assets	Composition	Evaluation

[RR03]	Experiences with software product family evolution	2003	Analyze and plan	Decision-making	SPL architecture	NA	Experience
[Sha99]	Exploiting object technology to support product variability	1999	Implement	Built-for-change	Code assets	Hybrid	Experience
[HRG12]	Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions	2012	Implement	Change synchronization	Variability model, Code assets	Model-driven	Evaluation
[RBK14]	Feature maintenance with emergent interfaces	2014	Analyze and plan	Change impact	Code assets	Annotation	Validation
			Implement	built-with-change			
[SSTS14]	Feature-context interfaces: Tailored programming interfaces for software product lines	2014	Implement	Built-with-change	Code assets	Composition	Evaluation
[YM12]	Fine-grained change impact analysis for component-based product families	2012	Analyze and plan	Change impact	Code assets	Composition	Validation
[JT11]	Flexible generators for software reuse and evolution	2011	Implement	Built-with-change	Products	Model-driven	Solution
[AJB+14]	Flexible product line engineering with virtual platform	2014	Implement	Change synchronization	Products	Clone	Conceptual
[LRZJ04]	Framed Aspects: supporting variability and configurability for AOP	2009	Implement	Built-for-change	Code assets	Hybrid	Solution
[Tab04]	Generalized release planning for product line architectures	2004	Analyze and plan	Planning	SPL architecture	NA	Experience
[TBC08]	Identifying and exploiting the similarities between rationale management and variability management	2008	Analyze and plan	Decision-making	Variability model	NA	Evaluation
[Ana09]	Increasing efficiency and effectiveness of software product line evolution: An infrastructure on top of configuration management	2009	Implement	Change synchronization	Variability model, Code assets, Products	Composition	Validation
[Böc05]	Innovation management for product line engineering organizations	2005	Identify	Monitoring the environment	NA	NA	Conceptual

[KB12]	Issue-based variability management	2012	Analyze and plan	Decision-making	Variability model	NA	Evaluation
[SB00]	Issues concerning variability in software product lines	2000	Implement	Built-for-change	SPL architecture	Hybrid	Experience
[Liv11]	Issues in software product line evolution: complex changes in variability models	2011	Analyze and plan	Change impact	Variability model	Annotation	Experience
[DRC13]	Language features for software evolution and aspect-oriented interfaces: An exploratory study	2013	Implement	Built-for-change	Code assets	Composition	Evaluation
[JBAC15]	Maintaining feature traceability with embedded annotations	2015	Implement	Change synchronization	Variability model	Clone	Validation
[APT12]	Managing and assessing the risk of component upgrades	2012	Analyze and plan	Decision-making	Code assets	Composition	Experience
[JZZZ08]	Maintaining software product lines: an industrial practice	2008	Analyze and plan	Change impact	Code assets	Composition	Experience
[RCC13]	Managing cloned variants : A Framework and experience	2013	Analyze and plan	Change impact	Products	Clone	Validation
			Implement	Change synchronization			
[CCS+12]	Managing evolution in software product lines: A model-checking perspective	2012	Verify	Scalable verification	Variability model, Code assets	Model-driven	Solution
[RKBC12]	Managing forked product variants	2012	Analyze and plan	Change impact	Product	Clone	Conceptual
			Implement	Change synchronization			
[CCJM12]	Model-based product line evolution: An incremental growing by extension	2012	Identify	Monitoring products	Products	Model-driven	Solution
[SPP+13]	Model-driven planning and monitoring of long-term software product line evolution	2013	Analyze and plan	Planning	Variability model	Model-driven	Solution
[PBD+12]	Model-driven support for product line evolution on feature level	2012	Analyze and plan	Planning	Variability model	Model-driven	Validation
[HH07]	Modeling product line architectures through change sets and relationships	2007	Implement	Built-with-change	SPL architecture	Composition	Solution
[HGR10]	Negotiation constellations in reactive product line evolution	2010	Analyze and plan	Decision-making	NA	NA	Conceptual
[MWB11]	On the problems with evolving egemin's software product line	2011	Implement	Change synchronization	SPL architecture	Composition	Experience

[FGFd14]	On the use of feature-oriented programming for evolving software product lines - A comparative study	2014	Implement	Built-for-change	Code assets	Hybrid	Evaluation
[Sch06b]	Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems	2006	Verify	Scalable verification	Variability model, SPL architecture	NA	Experience
[McV15]	Preliminary product line support in BitKeeper	2015	Implement	Change synchronization	Code assets	Clone	Solution
[SC11]	Preserving the exception handling design rules in software product line context: A practical approach	2011	Verify	Inconsistency checking	Code assets	Composition	Validation
[KSK08]	Product line variability with elastic components and test-driven development	2008	Implement	Built-with-change	Products	Composition	Solution
[CKM+08]	Providing feedback from application to family engineering: The product line planning game at the Testo AG	2008	Identify	Monitoring products	Products	NA	Experience
[TBK09]	Reasoning about edits to feature models	2009	Analyze and plan	Change impact	Variability model	NA	Validation
[DKZH12]	Recovering commit dependencies for selective code integration in software product line	2012	Implement	Change synchronization	Code assets	Composition	Validation
[SK14]	Reducing the verification cost of evolving product families using static analysis techniques	2014	Verify	Scalable verification	Variability model, code assets	Annotation	Validation
[vO02]	Roadmapping a product population architecture	2002	Analyze and plan	Planning	SPL architecture	Composition	Solution
[TBG15]	Safe evolution of product populations and multi product lines	2015	Verify	Inconsistency checking	Variability model code assets	NA	Solution
[SK08]	Scheduling product line features for effective roadmapping	2008	Analyze and plan	Planning	NA	NA	Experience
[HFG+10]	Simulating evolution in model-based product line engineering	2010	Analyze and plan	Decision-making	NA	NA	Validation
[TMN08]	Software Configuration Management for Product Derivation in Software Product Families	2008	Implement	Change synchronization	Variability model, Code assets,	Composition	Solution

					Product		
[RUQ+13]	SPLEMMA: A generic framework for controlled-evolution of software product lines	2013	Implement	Built-with-change	Variability model	NA	Solution
[LDSL07]	State-based modeling to support the evolution and maintenance of safety-critical software product lines	2007	Analyze and plan	Decision-making	NA	NA	Solution
[KMNL06]	Static evaluation of software architectures	2006	Implement	Built-with-change	SPL architecture	NA	Experience
			Verify	Inconsistency checking			
[DGRN10]	Structuring the modeling space and supporting evolution in software product line engineering	2010	Implement	Change synchronization	Variability model, Code assets	Model-driven	Evaluation
[JRG+12]	Supporting model maintenance in component-based product lines	2012	Verify	Inconsistency checking	Variability model, Code assets	Model-driven	Validation
[MBKM08]	Supporting the grow-and-prune model in software product lines evolution using clone detection	2008	Identify	Monitoring products	Products	Clone	Evaluation
[SPZZ10]	Synchronized architecture evolution in software product line using bidirectional transformation	2010	Implement	Change synchronization	SPL architecture, Products	Model-driven	Solution
[KC05]	Synchronizing cardinality-based feature models and their specializations	2005	Implement	Change synchronization	Variability model	NA	Solution
[SV02]	The economic impact of product line adoption and evolution	2002	Analyze and plan	Decision-making	NA	NA	Experience
[MW11]	Towards a solution for change impact analysis of software product line products	2011	Analyze and plan	Change impact	Variability model, SPL architecture, Code assets	Model-driven	Conceptual
[MD15]	Tunning Github for SPL development: branching models and operations for product engineers	2015	Implement	Change synchronization	Code assets, Products	Composition	Solution

[HRGL12]	Using regression testing to analyze the impact of changes to variability models on products	2012	Analyze and plan	Change impact	Variability model	Model-driven	Validation
[CGCS04]	Using simulation to facilitate the study of software product line evolution	2004	Analyze and plan	Decision-making	NA	NA	Solution
[DSB09]	Variability assessment in software product families	2009	Analyze and plan	Decision-making	Variability model	NA	Validation
[SK01]	Violatility analysis framework for product lines	2001	Identify	Monitoring customer	NA	NA	Solution
[MARC13]	Visualization and exploration of optimal variants in product line engineering	2013	Analyze and plan	Change impact	Variability model	NA	Validation

Table A.4 Primary study facet classification.

Appendix B

ETL at *CustomDIFF*

This Appendix provides the algorithms that describes the ETL process followed by *CustomDIFF*.

B.1 Algorithms for the ETL process

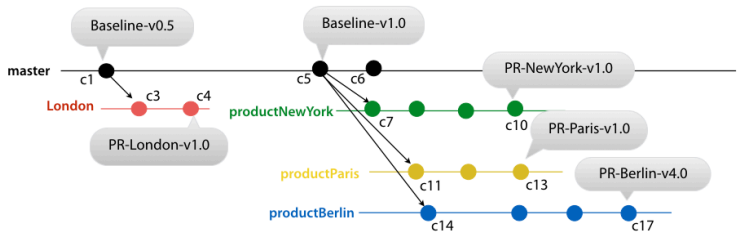

Figure B.1: *WeatherStationSPL* branching model: the *master* branch holds the core assets from where SPL products are branched off.

CustomDIFF's ETL follows traditional DW practices [KR02]. However, the extraction stage makes some assumptions about the underlying Git repository structure. The extraction process depends on the Git branching strategy being used. So far, *CustomDIFF* supports the branch-and-unite model [BP14]. Here, the *master* branch contains core-asset baseline releases while products branch off the *master* (see Figure B.1). Later on, product branches can be reunified with the *master* branch after releasing the product and pruning the branch.For automated processing, the following parameters need to be configured beforehand: (1) PR_PATTERN, i.e the pattern that product release tags should match (e.g. "PR-*"); (2) BASELINE_PATTERN, i.e the pattern that baseline release tags should match (e.g. "Baseline-*"), (3) VP_INIT_CLAUSE, i.e. the pattern that variation point opening clauses should match (e.g. "PV:IFCOND*"),

Algorithm B.1 Mining product customizations.

```
1  List<CustomizationFacts> MINE_CUSTOMIZATIONS (GitRepository
       gitRepo, String baseline_tag)
2
3  List<CustomizationFact> customization_facts = new List<
       CustomizationFact> ();
4  Commit baselineCommit = gitRepo.getCommitByTagName(baseline_tag
       );
5
6  List <Tag> all _tags = gitRepo.getAllTags()
7  for each tag in all_tags
8    Commit baseline;
9    if(tag.name.matches(PR_PATTERN)) do
10     baseline = getBaselineForRelease(tag);
11     if(baselineCommit === baseline) do
12       List<Diff> diffs = DIFF(baselineCommit, tag.getCommit())
13             for each diff in diffs
14           List<CustomizationFact> custom_facts =
                 extractCustomizationFacts (diff, tag);
15           customizations.add(custom_facts);
16         end_for_each
17     end_if
18   end_if
19 end_for_each
20 return customizations;
```

and (4) VP_END_CLAUSE, i.e. the pattern that product release tag should match (e.g "ENDCOND*"). Next, we delve into the details [1].

The process starts with the main function called *Mine_Customizations*[2], which takes a *GitRepository* and returns the set of *customization_facts* that have been performed to a given *baseline* by all the products derived from this baseline. Namely (line 1 in Algorithm B.1):

> List<Customization_Facts> Mine_Customizations (GitRepository gitRepo, String baseline_tag)
>
> where GITREPO stands for the git repository where the SPL is being developed; BASELINE_TAG stands for the name of the *git tag* that identifies the baseline for which the customization facts will be computed. To illustrate the algorithm with a running example, take the content of Figure B.1 as the GITREPO; "Baseline-v1.0" as the value for BASELINE_TAG, "PR-" as the value for

[1]Note that in Git, commits are chained with each other from "parent" to "child". This means that, although in Figure B.1 C5 was committed before C7, we can not reach C7 from C5, as C7 points to C5. Likewise, C5 points to C1, and C10 to C9. Keep this fact in mind when reading the next algorithm.

[2]This algorithm was implemented in Java, using the JGit library `http://www.eclipse.org/jgit/`

PR_PATTERN, "Baseline-" as the value for BASELINE_PATTERN, and "PV:INFOND" and "PV:ENDCOND" as the values for VP_INIT_CLAUSE and VP_END_CLAUSE, respectively. Algorithm B.1 provides the details:

1. Identify which is the BASELINECOMMIT to analyze (line 4). The function GETCOMMITBYTAGNAME returns the commit to which the BASELINE_TAG points to. For our running example BASELINECOMMIT holds the commit C5.

2. Identify the product releases that were derived from the BASELINECOMMIT (lines 6-11). This implies to:

   (a) From all the tags in GITREPO, identify those that are product releases (lines 6-9). First, collect all the existing tags in the repository (line 6). For our running example, the variable ALL_TAGS holds now: *London-v1*.0, *NewYork-v1*.0, *Paris-v1*.0, *Berlin-v4*.0, *Baseline-v0*.5 and *Baseline-v1*.0. Second, filter out those tags that are not product releases. i.e. those that do not match the PR_PATTERN (line 7-9). For our running example tags *Baseline-v0*.5, and *Baseline-v1*.0 are filtered out.

   (b) Filter out the product releases that were not actually derived from the BASELINECOMMIT (lines 10-11). This implies, first, to identify the baseline commit each product release was derived from. This is calculated by calling to the method GETBASELINEFORRELEASE (line 10), which takes a product release tag (e.g. *Berlin-v*4.0), traverses the git history (e.g backwards from C17) until it finds a commit tagged with a label that matches the pattern BASELINE_PATTERN (e.g. *Baseline-v1.0*), and finally, returns the commit it points to (e.g. C5). Second, filter out those product releases whose baseline is not equal to BASELINECOMMIT (line 11). For our running example, the product release *London-v1*.0 would be filtered out, as the BASELINE it was derived from is C3 instead of C5.

3. Finally, compute the customization facts for each product release that was indeed derived from BASELINECOMMIT (lines 12-16). This implies for each product release to:

   (a) Perform a *DIFF* operation between the BASELINECOMMIT and the commit to which the product release tag is pointing to (line 12). For instance, the *DIFF* operation for the product release tag *Berlin-v*4.0 would be as follows: DIFF(C5, C17). The result of the operation, i.e. DIFFS, is the list of diff-outputs (a.k.a patches), one per file that the product has changed from the baseline. For instance, if the product release *Berlin-v*4.*0* changes five files from the baseline, then DIFFS would contain five diff-output files, each per file changed (see Figure B.2 as an example of a diff-output).

   (b) For each diff-output, extract the customization facts by calling to the method EXTRACTCUSTOMIZATIONFACTS (line 14). This method, parses the diff-output, identifies the set of consecutive changes performed to the same variation point, and returns the corresponding customization facts (see Algorithm B.2 next).

```
diff --git a/input/js/sensors.js b/input/js/sensors.js
index 8409dfa..beaf743 100644
--- a/input/js/sensors.js
+++ b/input/js/sensors.js
@@ -26,12 +26,16 @@ function applyTachoValue(min, max, measureText, p
        var divisor = Math.round((max - min)/13);
        var c = Math.round(divisor/2);

+       var tmp= getTmpFomMeassure(measureText);
        // PV:IFCOND(pv:hasFeature('Temperature'))
        if (measureText && pointer) {
                var measure = measureText.value;
                var intValue = checkMeasure(min, max, measure);
                if (isNaN(intValue)) return false;
-
+               measureText = document.getElementById("w_measure");
+               windMeasure = measureText.value;
+                pointer2 = document.getElementById("w_point");
+
                intValue -= min;
                if (intValue % divisor < c) intValue -= intValue % di
                else intValue += divisor - intValue % divisor;
```

Figure B.2: The diff-output (a.k.a. patch) for the DIFF(C5, C17), w.r.t file sensors.js file.

(c) Finally, add the extracted customization facts to the global container CUSTOMIZATIONS (line 15), and when all product releases are mined, return this container (line 20).

How diff-outputs are read, and constructed, is important to understand our next algorithm. As a reminder of how a diff-output looks like, take Figure B.2, which shows the changes that product *Berlin* has performed to the file *sensors*.js. The first 4 lines give the details of the file being compared (i.e. sensors.js), and the rests are the *hunks*. When comparing two versions of the file, the *diff* operation tries to record differences as groups of differing lines, and uses common lines (context lines) to anchor these groups. Such groups are called *hunks* of difference and follow the pattern: @@ *old-file-range new-file-range* @@ *[heading]*. Note how the diff-output in Figure B.2 only contains one hunk. The *old-file-range* is in the form: -<*start*_line>,<number_of_*lines*>, and *new_file_range* is: +<*star_line*>,<*number_of_lines*>. *Start_line* and *number_of_lines* refer to the position and the hunk length in the original version the and new version, respectively. Therefore, the line "*var divisor = Meath.round (max -min)/13*" in Figure B.2, corresponds to the line 26 in the older version of the file (i.e the baseline version of *sensors.js*), and also corresponds to the line number 26 in the newer version of the file (i.e. the product Berlin version of *sensors.js*). Hence, if we would like to know line position for the change "+ var tmp = getTmpForMeassure(measureText)" in the new version of the file, we would need to: take the line number 26 (the first line in the hunk), and sum the number of context lines (+ 3) and the number of added lines (+1) until we reach the line "+ var tmp = getTmpForMeassure(measureText)".

The *extractCustomizationFacts* algorithm takes as input a diff-output (take the example in Figure B.2), and extracts a set of customization facts, i.e. the consecutive changes made to a given variation point. Namely:

Algorithm B.2 Extracting customization facts from a diff-output file.

```
List<CustomizationFact> extractCustomizationFacts(String diff,
    Tag tag)

List<CustomizationFact> customizations = new List<
    CustomizationFact> ();
String fileName = extractFileNameFromDiff(diff);
List<String> hunks = diff.split("@@");
for each hunk in hunks
  List<String> lines = hunk.split("\n");
  String custom_diff ="";
  int added_lines = 0, deleted_lines =0, context_lines = 0;
  VariationPoint vp;
  int lineNumberOld = extractLineNumberFromHunk(lines.get(1));
  int lineNumberNew = extractLineNumberFromHunk(lines.get(1));
  for each line in lines
    customDiff.concat.(line);
  if(line.startsWith("+") AND (!line.contains(VP_INIT_CLAUSE))
      AND (!line.contains(VP_END_CLAUSE))) //added line
      identified
     added_lines ++;
  else if(line.startsWith("-") AND (!line.contains(
      VP_INIT_CLAUSE)) AND (!line.contains(VP_END_CLAUSE)))  //
      deleted line identified
  deleted_lines ++;
 else if(line.startsWith(" ") AND (!line.contains(
     VP_INIT_CLAUSE)) AND (!line.contains(VP_END_CLAUSE))) //
     context line
    context_lines++;
    else if (line.contains(VP_INIT_CLAUSE) OR (line.contains(
        VP_END_CLAUSE))){
       vp = extractVpFromFileAndLine(filename, lineNumberNew -1
            + added_lines + context_lines);
       customDiff = fixHeaderForCustomDiff(customDiff, vp.
           getExpression(), lineNumberOld, lineNumberNew);
       customDiff = customDiff.concat(line+1).contact(line+1);
           //add context lines
       CustomizationFact cust = new CustomizationFact (
           customDiff, added_lines, deleted_lines, vp, tag.name)
           ;
       customizations.add(cust);
       lineNumberOld = lineNumberOld + context_lines +
           deleted_lines;
       lineNumberNew = lineNumberNew + context_lines +
           added_lines;
       deleted_lines = 0; added_lines = 0; context_lines = 0;
    end_else_if
   end_for_each
  if (vp== null) //no vp was found
    vp = extractVpFromFileAndLine(filename, lineNumberNew)
  hunk = fixHeaderForCustomDiff(hunk, vp.expression,
      lineNumberOld, lineNumberNew);
  CustomizationFact cust = new CustomizationFact (hunk,
      added_lines, deleted_lines, vp, tag.name);
    customizations.add(cust);
  end_if
end_for_each
return customizations;
```

LIST<CUSTOMIZATION_FACTS> EXTRACTCUSTOMIZATIONFACTS (DIFF DIFF, TAG PR)

where DIFF stands for the diff-output from which to extract the customization facts from; and TAG stands for the product release tag for which the customizations are being computed. As the running example, take the content of Figure B.2 as the value for the DIFF, and "PR-Berlin-v4.0" as the value for the TAG. Algorithm B.2 provides the details:

1. Elucidate which is the file name being diff-ed (line 3). This implies calling to the method EXTRACTFILENAMEFROMDIFF, which takes as input a diff-output (i.e DIFF), and extracts the FILENAME. For our running example, the EXTRACTFILENAMEFROMDIFF would return *sensors.js*.

2. Identify the set of consecutive changed lines that correspond to the same variation point, and create the corresponding customization facts (lines 14-35). Namely:

   (a) Traverse each HUNK line by line. This requires to:

      i. Split the DIFF into HUNKS, and convert each HUNK into a list of LINES (lines 5-7). For our running example there is only one hunk, and the LINES would be those in Figure B.2, excluding the first four.

      ii. Initialize the variables that support the computation of a customization fact (lines 7-12): the additions, deletions and context lines (i.e. ADDED_LINES, DELETED_LINES, CONTEXT_LINES, respectively), the new diff-output, based on the variation point being affected by the changes (i.e. CUSTOM_DIFF), the actual variation point (i.e. VP), and the range information to build the diff correctly (i.e. LINENUMBEROLD, LINENUMBERNEW).

   (b) Until a LINE containing a VP_INIT_CLAUSE or VP_END_CLAUSE is not identified, increment the corresponding counters when an addition, deletion, or contextual line is found (lines 16-21).

   (c) Create a new customization fact when an LINE containing a VP_INIT_CLAUSE or VP_END_CLAUSE is identified (lines 23-34). For our running example, say LINE is equal to "PV:IFCOND (pv:hasFeature("Temperature'))". Namely:

      i. Elucidate to which variation point do the already traversed lines belong to (those lines prior to the VP_INIT_CLAUSE). The method EXTRACTVPFROMFILEANDLINE takes a FILENAME and a line number, and returns the variation point to which the line number belongs to (line 24). The method would return that the previous line, i.e. "+ var tmp = getTmpForMeasureText", belongs to a variation point which expression is (WindSpeed or AirPressure).

      ii. Build up the new diff for the set of consecutive changed lines (i.e. "+ var tmp = getTmpForMeasureText"). The variable CUSTOM_DIFF

```
diff --git a/input/js/sensors.js b/input/js/sensors.js
index 8409dfa..beaf743 100644
--- a/input/js/sensors.js
+++ b/input/js/sensors.js
@@ -26,6 +26,7 @@ Expression: hasFeature(WindSpeed or AirPressure)
        var divisor = Math.round((max - min)/13);
        var c = Math.round(divisor/2);

+       var tmp= getTmpFomMeassure(measureText);
        // PV:IFCOND(pv:hasFeature('Temperature'))
        if (measureText && pointer) {
                var measure = measureText.value;
```

```
diff --git a/input/js/sensors.js b/input/js/sensors.js
index 8409dfa..beaf743 100644
--- a/input/js/sensors.js
+++ b/input/js/sensors.js
@@ -29,9 +30,22 @@ Expression: hasFeature (Temperature) -> nested into -> hasFeature(WindSpeed or AirPressure)
 // PV:IFCOND(pv:hasFeature('Temperature'))
        if (measureText && pointer) {
                var measure = measureText.value;
                var intValue = checkMeasure(min, max, measure);
                if (isNaN(intValue)) return false;
-
+               measureText = document.getElementById("w_measure");
+               windMeasure = measureText.value;
+                pointer2 = document.getElementById("w_point");
+
                intValue -= min;
                if (intValue % divisor < c) intValue -= intValue % divisor;
                else intValue += divisor - intValue % divisor;
```

Figure B.3: *Custom_diffs* obtained after applying Algorithm B.2 to the diff-output in Figure B.2: VP-1 (top) and VP-2 (bottom).

has been recording each traversed line up to the new variation point (line 15), but needs additional fixes. First, it needs two more contextual lines (line 26). Second, the hunk header needs to be changed: the range needs to be fixed, and the heading needs to state the expression to which the changed lines affect. The method FIXHEADERFORCUSTOMDIFF would do both. Figure B.2 (top) depicts the content of the CUSTOM_DIFF after being fixed.

iii. Create the new customization fact and add it to the total list of customizations (lines 28-29).

iv. Finally, reset the variables for the next customization fact to be identified (lines 31-33).

(d) If no variation point was identified during the analysis of the hunk, elucidate to which variation point does the DIFF content corresponds to, and create a customization fact accordingly (lines 36-41).

3. When all the hunks are analyzed, and hence, all customizations facts are computed, return the total list of customizations (line 39).

Figure B.2 depicts the two CUSTOM_DIFF derived from the main DIFF (depicted in Figure B.2) after applying *extractCustomizationFacts.*

Appendix C

A brief on git

This Appendix provides a bite on Version Control Systems (VCSs) and git basics.

C.1 Version Control Systems

Version Control Systems (VCSs), a.k.a. revision or source control systems, are tools that support concurrent and collaborative software processes by: (1) seamlessly tracking changes to the source code, and (2) letting multiple developers collaborate efficiently. A VCS repository stores of all the files under version control, as well as their previous versions. Developers, in order to work on the source code, check-out a version of the files to a local workspace. Developers make changes to local files on their workspace, and **commit** (a.k.a check-in) to make permanent software changes to the central repository. Commits are chained together, with each new version committed to the repository. Over time, a sequence of changes is represented as a series of commits, known as the repository history. **Branches** are used to launch a separate lines of development, and allow the development to continue in multiple directions simultaneously, without interfering into each others work. Eventually, a branch is **merged** (fused) with other branches to reunite disparate efforts, usually, by a three-way merge. As a result, a new version is created. When this process does not go smoothly (i.e. different changes where made to the same part of the same file), the user has to manually resolve the conflicts.

VCSs can broadly be classified into centralized and distributed. Centralized VCSs (CVCS) came first in history. In a CVCS there is a single central repository to which which clients synchronize. The local workspace of the clients only holds a copy of the files that reside in the server. When any CVCS operation needs to be performed, e.g committing the changes from the local workspace to the repository, review the history a file, branch, merge, etc., clients need to connect to the server. Conversely, in a distributed VCS (DCVS) clients don't just checkout the latest version of the files, but they get a full-fledged repository: with the whole history and the power to exchange source code changes with other repositories, i.e. peer-to-peer.

DVCS introduce three additional operations to VCSs: **clone**, **push** and **pull**. A

clone, copies a repository into a developers machine. Indeed, in DVCSs each developer has each own local repository. A **clone**, is ideally made only once, i.e. when a developer joints the project for the first time. Afterwards, traditional VCSs operations (e.g. commit, branch, merge) are performed against developer's local repository. To sync with other peers, developers would conduct a **push** (publishing local changes to another repository) and **pull** (getting changes from a remote repository to a local one) operations. A peer-to-peer collaboration approach opens new collaboration workflows not previously possible with centralized VCSs[1]. Specially, the *fork&pull* model has been beneficial Open Source Software projects. Nevertheless, DVCS can mimic a centralized model if developers sync to a canonical repository and they don't sync between them.

C.2 A brief on Git and GitHub

Git is a DVCS. The Eclipse Foundation reported in its annual community survey that as of May 2014, Git is now the most widely used source-code management tool, with 42.9% of professional software developers reporting that they use Git as their primary source control system[2]. This figure grounds the selection of Git in this Thesis. This section outlines the main operations supported by Git (i.e. *commit, branch* and *merge*) and its web counterpart, GitHub (i.e. *fork* and *pullRequest*).

C.2.1 Data Structures: the Git Object Model

The major difference between *Git* and other *VCSs (e.g. CVS*, *Subversion*) is the way it stores data: as a set of snapshots of a mini filesystem instead of a set of changes. Every time a client *saves* (i.e. commits) the state of the project, Git takes a snapshot of all the files. *Git* includes four objects: TREE, BLOB, COMMIT and TAG (see Figure C.1) which are characterized along the following properties: *sha* (unique *hash* identifier based on the objects' content), *type* (*"tree"*, *"blob"*, *"commit"* or *"tag*"), *content* and *size* (in bytes). TREE objects, represent file-system directories, which can contain further TREEs and BLOBs. BLOB objects, represent a file storing data. COMMIT objects, represent a version (i.e. snapshot) for the project at a certain time. COMMIT's *project* attribute, hold the top-level directory for the project artefacts. Commit objects, are preceded by its antecesor commit (zero *parents* for the first commit in the commit history, one *parent* for a normal commit, and multiple parents for a commit that results from a merge of two or more branches). Commit objects, contain additional metadata: *message* (user entered message describing the changes), *commiter* (the user who performs the commit), *author* (the user responsible for the change) and *time* (the moment on which the commit was performed). Note, that a COMMIT object does not itself contain any information about what was actually changed. All changes are calculated by comparing the contents of the project *tree* referred to by the commit, with the *trees* associated with its parent. Finally, TAG objects serve to tag special commits, such as, releases, hot-fixes, etc. A *Git* REPOSITORY, comprises

[1] https://tinyurl.com/jb37na6

[2] https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/

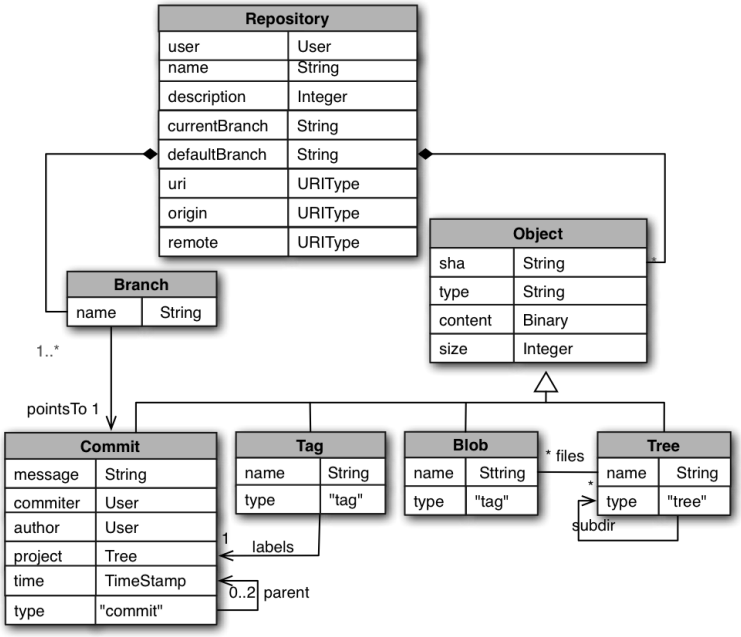

Figure C.1: Git Object Model

all objects aforementioned. Additionally, *Git* repositories, have BRANCHEs, which is just a lightweight movable pointer to a COMMIT. *Git* repositories must have at least one branch (i.e. the *default branch*). The default branch, is destined for the main-line of development. When a *Git* repository is created, *Git* automatically creates a branch named *master*, and sets it as the repository's default branch. *Git* repositories further include: repository owner *user*, repository *name*, a little *description* about the repository project, the *currentBranch* the user is working at, the *uri* of the repository, a *origin*, if the repository is a clone of another Git the repository, and the *remote* repository to which it synchronizes (if any).

C.2.2 Git Basic Operations

Commit. A commit operation makes permanent the software changes to a repository. Git, converts the users' working directory into a Git snapshot, i.e. $\text{COMMIT}_{\text{Object}}$(see Figure C.2). When a commit operations is performed, the branch moves forward automatically, pointing to the last commit performed. This way, commit objects are chained together, with each new snapshot pointing to its predecessor (*c2* points to *c1*). Over time, a sequence of changes is represented as a series of commit objects, known as the repository *history*. The operation, can be implicitly described as, COMMIT: USER X REPOSITORY X BRANCH X TREE X STRING -> $\text{COMMIT}_{\textbf{OBJECT}}$[3].

[3]We distinguish commit operation and commit object as, COMMIT and $\text{COMMIT}_{\text{OBJECT}}$, respectively.

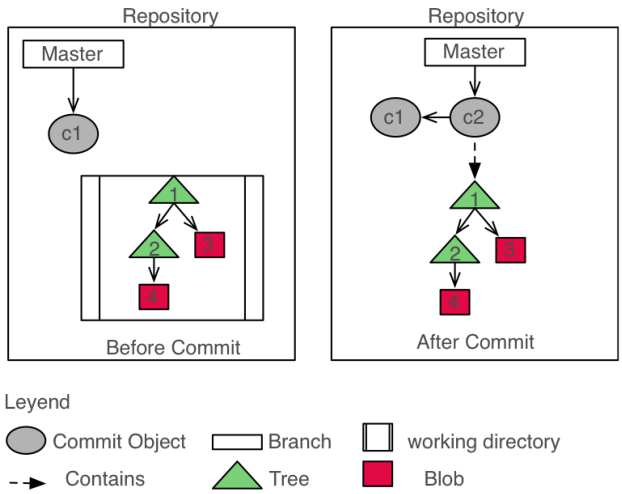

Figure C.2: Commit operation

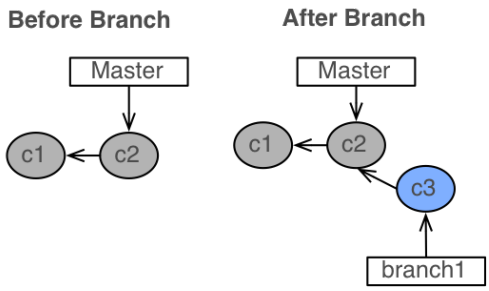

Figure C.3: Branch + Commit Operation

Branch. It launches a separate line of development. Branches are created upon an existing *commit* (users must set first the branch-to-be name). Figure C.3, depicts the repository before and after performing a branch operation to create *branch1* upon *master*. When branch operation is performed, *branch1* points to *c2* commit object. Further commits to *branch1* (i.e. commit *c3*), make the two branches diverge. The branch operation can be described as BRANCH: REPOSITORY x BRANCH x IDENTIFIER -> BRANCH.

Merge. Often, a branch is *fused* (i.e. merged) with other branches to reunite disparate efforts. Figure C.4, depicts a merge operation scenario, where the *headBranch* will be merged into the *baseBranch.* Git does a simple three-way merge, using the two commit objects pointed to by branches *c3* and *c2*, and their common ancestor (*i.e. c1*). As a result, a new commit, *c4*, is created for *baseBranch*. When this process does not go smoothly (i.e. different changes where made to the same part of the

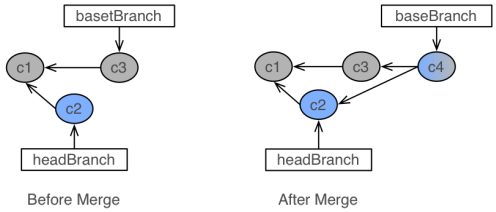

Figure C.4: Merge Operation

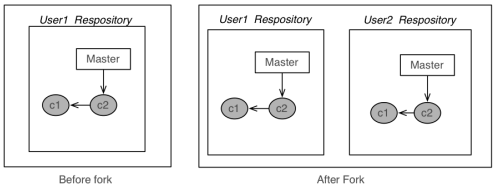

Figure C.5: Fork Operation

same file), Git adds standard conflict-resolution markers to the files that have conflicts, so that users can detect them, and resolve them manually. The merge operation can be defined as: MERGE: User X Repository X Branch X Branch -> $\text{COMMIT}_{\text{OBJECT}}$.

GitHub[4] is the largest open source Git repository hosting service. GitHub provides a Web-based graphical interface, and introduces social functionalities that make developer's identity and activities visible to other users [ref -Social Coding in GitHub]. This is particularly interesting for SPLs where two different teams need to collaborate: domain engineers and application engineers. GitHub introduces two new operations: *fork* and *pull request*.

Fork. This is the mechanism to make copies (i.e. *clones*) of entire repositories across *GitHub* users. When a user clicks on the *fork* button, *GitHub* automatically copies that repository to the user who requests the fork. This operation keeps both repositories connected, the source repository knows all its *forks* repositories, while the *forked* repository sets its *origin* attribute with the source repository *uri* . Fork operation is defined as, FORK: REPOSITORY X USER -> REPOSITORY.

Pull request. Whenever one of the users thinks his changes to the repository are appropriate for the other party, he can send him back as form of a pull request. A *pull request* operation, is basically a *merge* request between two repository branches. The user sending the request must specify the following attributes: (*i*) the *base repository* and *base branch*, where changes should be applied, (*ii*) the *head repository* and the *head branch*, meaning what changes the user like to apply, and (*iii*) a *message*, describing the proposed changes. Changing the base repository branch, changes who is notified of the pull request (the base repository gets the request). Once a pull request is sent, interested parties can review the set of changes, discuss potential modifications, and even push follow-up commits if necessary. *A* pull request operation

[4]https://github.com/

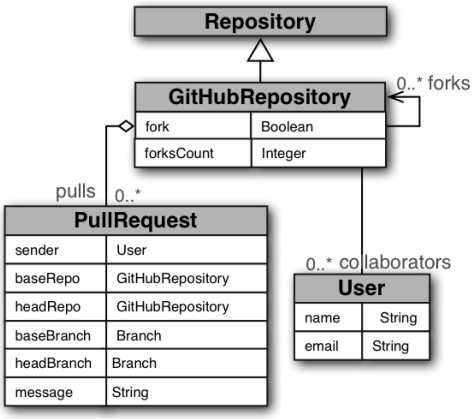

Figure C.6: *GitHub* additional object Model (partial model).

can be described as: CREATEPULLREQUEST: USER x REPOSITORY x REPOSITORY x BRANCH x BRANCH ->PULLREQUEST. Both operations, require Git to handle additional objects, which are depicted in Figure C.6. This model of collaboration is commonly called the *fork&pull* model, which is very popular for the development of open-source projects.

C.3 Branching models in VCSs

A branching model embodies the rationales adopted for branching and merging configuration items within a VCS [WS02b]. It closely matches a team's software development process: it tells (1) how developers develop and collaborate with each other for new development, (2) how engineers release software both to test department and customers, and (3) how they deal with production fixes, i.e bugs that occur to the software released to customers. There is no a one-fit-all branching model, and each team needs to find its own.

Inherent to branching models there is the notion of the integration branch (a.k.a. mainline or trunk), where all developers integrate their changes to collaborate. The integration branch holds the current state of the project, and it is usually from where automated or nightly builds are triggered. Additionally, dedicated *feature* branches, i.e. auxiliary branches that branch from the integration branch, might be used to develop codebase changes without interfering other teammates. These branches should be as short-lived as possible to avoid deviating to much from the mainline and avoid merge hell. Finally, release branches serve to prepare a software release to distribute it outside the development activity (e.g. to customers or QA testing). It might hold also to small bug-fixes prior to the actual software release. The final version produced after QA that is released for production is usually tagged within the release branch (so that it can be identified for future needs). However, some others prefer to merge it to release

(a.k.a production) branches, which only holds the sequential versions of the software released to customers. These different purposes call for different branches within the VCS project.

Next, we describe four branching models though for single-systems, each of which is suitable for a different development practice.

Git-flow [Gitb] supports *formal releases* on a longer term interval (a few weeks to a few months) [cha] (see Figure C.7-B). For parallel development *feature* branches are used, i.e. short-lived branches aimed to develop new functionality without interfering with other teammates work. Collaboration between all developers is achieved when *feature* branches are merged to *develop*, i.e. the integration branch (from where nightly build should be triggered). When all the features planned for the next release have reached the integration branch, a release branch is created for testing and bug-fixing all products before is sent to production. When it reaches the desired quality, it is merged to *master*, i.e. the release branch that holds all sequential versions of the core assets from where products are released to customer, and tagged. Production bugs are solved through *hotfix* branches.

Trunk-based [tru] supports *continuous integration* (CI), i.e. *"a software development practice where members of a team integrate at least daily*" [CI-]. There are no parallel branches for development (see Figure C.7-a). Instead, all developers work together in a single branch, i.e. the *integration* branch [5]. The only branches permitted are release branches, used for testing the software. If during such testing bugs are identified these are worked in the integration branch and cherry-picked to the release branch. When the software is stable is it tagged and send to production. To resolve any production bug a similar approach is conducted. Instead, others advocate to also work the defects within the release branch and cherry-pick them to the integration branch (e.g. [bar]).

CoDe:U [CoD] supports a *continuous delivery* (CoDe) [HF10] approach (see Figure C.7-D). Based on git-flow, CoDe:U automates all merges between branches, by an *automated delivery pipeline* [HF10] , which covers all required steps, e.g retrieving code from the repository, building binaries and running tests. The single branches engineers can use for development and bug-fixing are *feature branches*. The rest is automated by *pipeline* that will: (1) merge commits from *feature* branch to the *integration* branch only if the result is a build-able state, (2) merge integration to *the stable* if it passes, e.g., regression tests, and (3) merge to *the release* branch if, e.g., the commit in stable is delivered to production. To deal with production fixes, *maintenance* branches are used.

Github-flow [gita] supports *continuous deployment* (CoDep) practice, specially targeted for web applications. CoDep extends CoDe by deploying to production every change committed to the integration branch (upon success of build and requires automated tests). Hence, anything in the integration branch is

[5] Teams might use *feature toggles* and *branch by abstraction* to disable uncompleted features.

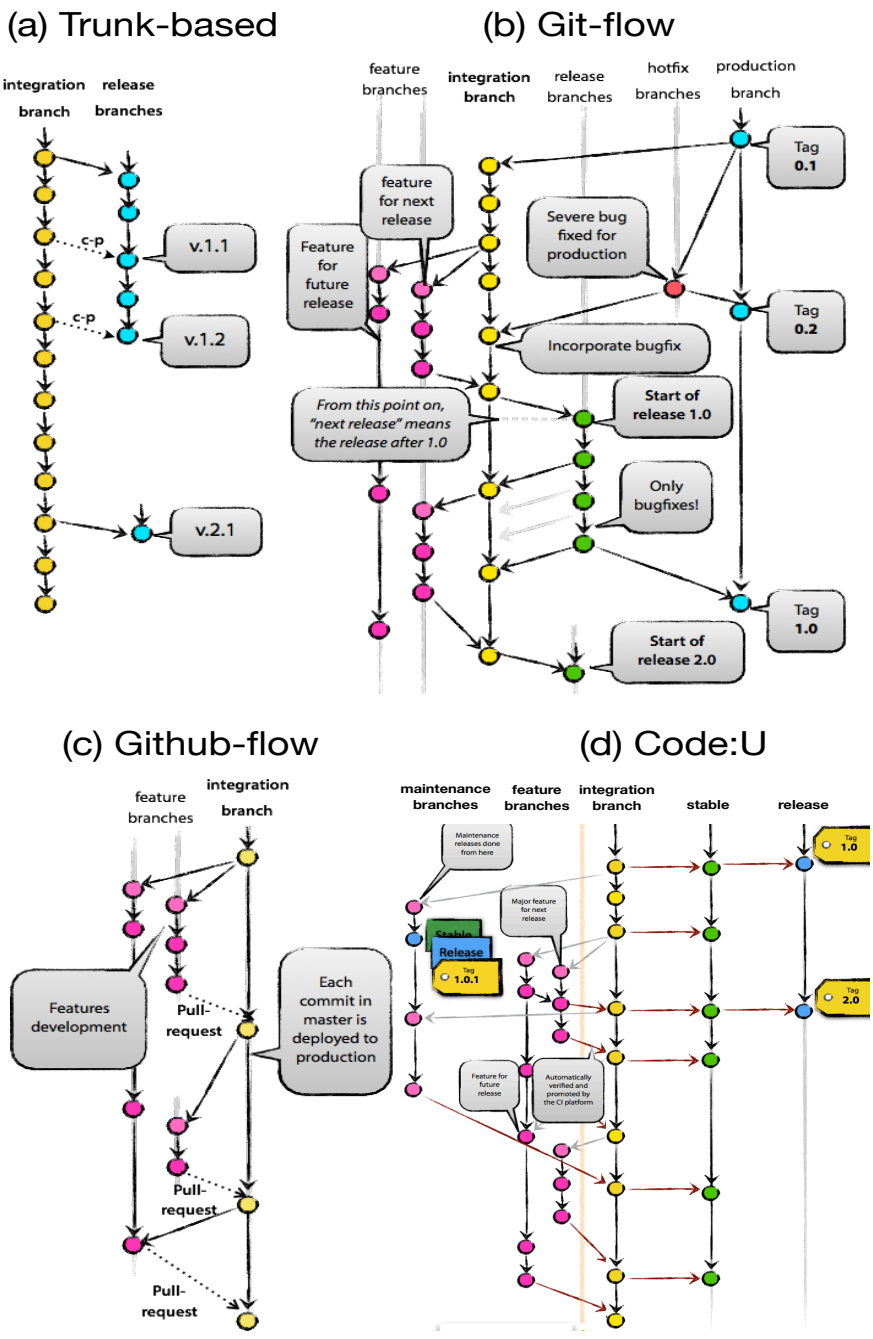

Figure C.7: Branching models for CPF (single-systems).

deployable (see Figure C.7-C). Github-flow uses *feature* branches for parallel development. When developers are ready to integrate *feature* branches they will issue a pull-requests[6] (PR). The PR will be reviewed by other teammates and signed off. Additionally, a build job would check wether the resulting merge of the PR is build-able. Only when the PR passes the review and the build, it is merged to the integration branch and deployed to production. To solve a production bug the same process needs to be carried out. No release branches are used, neither tags to identify releases.

[6] https://help.github.com/articles/about-pull-requests/

Bibliography

[3waa] Panlo santos. Three-way merging: A look under the hood. http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902. Accessed: 2018-03-26.

[3wab] Slant.co. What are the best visual merge tools for git? https://www.slant.co/topics/48/~best-visual-merge-tools-for-git. Accessed: 2018-03-26.

[ABCO98] Brad Appleton, S Berczuk, R Cabrera, and R Orenstein. Streamed Lines : Branching Patterns for Parallel Software Development. *PLoP conf.*, 98:98–25, 1998.

[ABKS13a] Sven Apel, Don S Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[ABKS13b] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[AC07] Sangim Ahn and Kiwon Chong. Requirements Change Management on Feature-Oriented Requirements Tracing. In *Computational Science and Its Applications - {ICCSA} 2007, International Conference, Kuala Lumpur, Malaysia, August 26-29, 2007. Proceedings, Part {II}*, pages 296–307, 2007.

[ACA08] Vander Alves, Tarcisio Camara, and Carina Frota Alves. Experiences with mobile games product line development at meantime. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 287–296, 2008.

[AD07] Samuel Ajila and Razvan T. Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80(1):74–91, 2007.

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. Refactoring product lines. In *Generative Programming and Component Engineering, 5th International Conference, {GPCE} 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, pages 201–210, 2006.

[AJB+14] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lammel, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. Flexible product line engineering with a virtual platform. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 532–535, 2014.

[AK08] Samuel A Ajila and Ali B Kaba. Evolution support mechanisms for software product line process. *Journal of Systems and Software*, 81(10):1784–1801, 2008.

[AKEs12] Walid Abdelmoez, Hatem Khater, and Noha El-shoafy. Comparing maintainability evolution of object-oriented and aspect-oriented software product lines. In *18th International Conference on informatics and Systems (INFOS 2012)*, pages 53–60, 2012.

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Trans. Software Eng.*, 39(1):63–79, 2013.

[AKM+10] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. A model-driven traceability framework for software product lines. *Software and System Modeling*, 9(4):427–451, 2010.

[All] Visualizing categorical data as flows with alluvial diagrams. `http://digitalsplashmedia.com/2014/06/visualizing-categorical-data-as-flows-with-alluvial-diagrams/`. Accessed: 2018-03-26.

[Ana09] Michail Anastasopoulos. Increasing Efficiency and Effectiveness of Software Product Line Evolution: An Infrastructure on Top of Configuration Management. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 47–56, New York, NY, USA, 2009. ACM.

[Ana13] Michail Anastasopoulos. *Evolution Control for Software Product Lines: An Automation Layer over Configuration Management*. PhD thesis, Fraunhofer IESE, 2013.

[and82] Fred K. Weigel and. Innovation characteristics and innovation adoption-implementation: A meta-analysis of findings. In *IEEE Transactions on Engineering Management*, pages 28–45, 1982.

[ANS+04] Walid Abdelmoez, Diaa Eldin M Nassar, Mark Shereshevsky, Nicholay Gradetsky, Rajesh Gunnalan, Hany H Ammar, Bo Yu, and Ali Mili. Error Propagation In Software Architectures. In *10th {IEEE} International Software Metrics Symposium {(METRICS} 2004), 11-17 September 2004, Chicago, IL, {USA}*, pages 384–393, 2004.

[AP98] Ritu Agarwal and Jayesh Prasad. A conceptual and operational definition of personal innovativeness in the domain of information technology. *Information Systems Research*, 9(2):204–215, 1998.

[APT12] Maria Carmela Annosi, Massimiliano Di Penta, and Genny Tortora. Managing and assessing the risk of component upgrades. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering, {PLEASE} 2012, Zurich, Switzerland, June 4, 2012*, pages 9–12, 2012.

[AYD13] Nur Hani Zulkifli Abai, Jamaiah H. Yahaya, and Aziz Deraman. User requirement analysis in data warehouse design: A review. *Procedia Technology*, 11(Supplement C):801 – 806, 2013. 4th International Conference on Electrical Engineering and Informatics, ICEEI 2013.

[bar] https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/.

[BB11] Joerg Bartholdt and Detlef Becker. Re-engineering of a hierarchical product line. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 232–240, 2011.

[BBHL94] Barry W Boehm, Prasanta K Bose, Ellis Horowitz, and Ming June Lee. Software requirements as negotiated win conditions. In *Proceedings of the First {IEEE} International Conference on Requirements Engineering, {ICRE} '94, Colorado Springs, Colorado, USA, April 18-21, 1994*, pages 74–83, 1994.

[BCC98] Sergey Berezin, Sergio Campos, and EdmundM. Clarke. Compositional Reasoning in Model Checking. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer Berlin Heidelberg, 1998.

[BGvS10] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. Defining and Applying Detection Strategies for Aspect-Oriented

Code Smells. In *24th Brazilian Symposium on Software Engineering, {SBES} 2010, Salvador, Bahia, Brazil, September 27 - October 1, 2010*, pages 60–69, 2010.

[Bla02] A. F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 2–10, 2002.

[BLL08] Hongyu Pei Breivold, Stig Larsson, and Rikard Land. Migrating industrial systems towards software product lines: Experiences and observations through case studies. In *34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2008, September 3-5, 2008, Parma, Italy*, pages 232–239, 2008.

[BM14] Jorge Barreiros and Ana Moreira. A cover-based approach for configuration repair. In *18th International Software Product Line Conference, {SPLC} '14, Florence, Italy, September 15-19, 2014*, pages 157–166, 2014.

[Böc05] Günter Böckle. Innovation Management for Product Line Engineering Organizations. In *Software Product Lines, 9th International Conference, {SPLC} 2005, Rennes, France, September 26-29, 2005, Proceedings*, pages 124–134, 2005.

[Boh96] Shawn A Bohner. Impact analysis in the software change process: a year 2000 perspective. In *1996 International Conference on Software Maintenance {(ICSM} '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*, pages 42–51, 1996.

[Bos01] Jan Bosch. Software product lines: Organizational alternatives. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 91–100, 2001.

[Bos02] Jan Bosch. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In *Software Product Lines, Second International Conference, {SPLC} 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, pages 257–271, 2002.

[BP14] Goetz Botterweck and Andreas Pleuss. Evolution of Software Product Lines. In *Evolving Software Systems*, pages 265–295. 2014.

[BPPK09] Goetz Botterweck, Andreas Pleuss, Andreas Polzer, and Stefan Kowalewski. Towards feature-driven planning of product-line evolution. In *Proceedings of the First International Workshop on Feature-Oriented Software Development, {FOSD} 2009, Denver, Colorado, USA, October 6, 2009*, pages 109–116, 2009.

[BPS+12] Sebastian Barney, Kai Petersen, Mikael Svahnberg, Aybüke Aurum, and Hamish T. Barney. Software quality trade-offs: A systematic map. *Information & Software Technology*, 54(7):651–662, 2012.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[BR00] Keith H Bennett and Václav Rajlich. Software maintenance and evolution: a roadmap. In *22nd International Conference on on Software Engineering, Future of Software Engineering Track, {ICSE} 2000, Limerick Ireland, June 4-11, 2000.*, pages 73–87, 2000.

[Bre] Danilo Breuche. Product line engineering. https://productlines.wordpress.com/2011/12/20/strategies-for-releases-development-and-maintenance-in-product-line-part-2-release-and-maintenance/ Last accessed: 1 March, 2018.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *International Conference on Software Engineering (ICSE).*, 2003.

[BTBK08] David Budgen, Mark Turner, Pearl Brereton, and Barbara Kitchenham. Using mapping studies in software engineering. In *Proceedings of Psychology of Programming Interest Group (PPIG)*, volume 8, pages 195–204, 2008.

[BTG12] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. *Theor. Comput. Sci.*, 455:2–30, 2012.

[CB11] Lianping Chen and Muhammad Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information & Software Technology*, 53(4):344–362, 2011.

[CBT+14] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014.

[CCG+03] Ping Chen, Matt Critchlow, Akash Garg, Christopher van der Westhuizen, and André van der Hoek. Differencing and Merging within an Evolving Product Line Architecture. In *Software Product-Family Engineering, 5th International Workshop, {PFE} 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, pages 269–281, 2003.

[CCJM12] Stephen Creff, Joël Champeau, Jean-Marc Jézéquel, and Arnaud Monégier. Model-based product line evolution: an incremental

growing by extension. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 107–114, 2012.

[CCS+12] Maxime Cordy, Andreas Classen, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Managing evolution in software product lines: a model-checking perspective. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 183–191, 2012.

[CDG+12] Bruno Barbieri Pontes Cafeo, Francisco Dantas, Alessandro Cavalcante Gurgel, Everton T Guimarães, Elder Cirilo, Alessandro F Garcia, and Carlos José Pereira de Lucena. Analysing the Impact of Feature Dependency Implementation on Product Line Stability: An Exploratory Study. In *26th Brazilian Symposium on Software Engineering, {SBES} 2012, Natal, Brazil, September 23-28, 2012*, pages 141–150, 2012.

[CdOW11] Chessman K F Corrêa, Toacy Cavalcante de Oliveira, and Cláudia Maria Lima Werner. An analysis of change operations to achieve consistency in model-driven software product lines. In *Software Product Lines - 15th International Conference, {SPLC} 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*, page 24, 2011.

[CGCS04] Yu Chen, Gerald C Gannod, James S Collofello, and Hessam S Sarjoughian. Using Simulation to Facilitate the Study of Software Product Line Evolution. In *7th International Workshop on Principles of Software Evolution {(IWPSE} 2004), 6-7 September 2004, Kyoto, Japan*, pages 103–112, 2004.

[CGM14] Sven Casteleyn, Irene Garrigós, and Jose-Norberto Mazón. Ten years of rich internet applications: A systematic mapping study, and beyond. *TWEB*, 8(3):18:1–18:46, 2014.

[cha] Scott chacon's blog. http://scottchacon.com/2011/08/31/github-flow.html.

[CI-] Martin fowler blog. https://www.martinfowler.com/articles/continuousintegration.html.

[CKM+08] Ralf Carbon, Jens Knodel, Dirk Muthig, Gerald Meier, and Testo Ag. Providing Feedback from Application to Family Engineering - The Product Line Planning Game at the Testo AG. *Proceeding of the 12th International Software Product Line Conference (SPLC)*, (01):180–189, 2008.

[CL01] Richard Cardone and Calvin Lin. Comparing Frameworks and Layered Refinement. In *Proceedings of the 23rd International Conference on Software Engineering, {ICSE} 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 285–294, 2001.

[CN01a] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[CN01b] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CNLL15] Fabio Calefato, Roberto De Nicolò, Filippo Lanubile, and Fabrizio Lippolis. Product Line Engineering for NGO Projects. pages 1–4, 2015.

[CoD] An automated git branching model. http://www.josra.org/blog/an-automated-git-branching-strategy.html.

[DA15] Oscar Díaz and Cristobal Arellano. The augmented web: Rationales, opportunities and challenges on browser-side transcoding. *To appear at ACM Transactions on the Web*, 2015.

[Dan] Danfoss drives in the product line hall of fame. `http://splc.net/hall-of-fame/danfoss-drives/`. Accessed: 2018-02-23.

[Dav89] Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Q.*, 13(3):319–340, September 1989.

[DCB09] Benjamin Delaware, William R Cook, and Don Batory. Fitting the Pieces Together: A Machine-checked Model of Safe Composition. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 243–252, New York, NY, USA, 2009. ACM.

[dCM+11] Paulo Anselmo da Mota Silveira Neto, Ivan Do Carmo Machado, John D McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, may 2011.

[DD08] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information & Software Technology*, 50(9-10):833–859, 2008.

[ddC+12] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, Yguaratã Cerqueira Cavalcanti, Eduardo Santana de Almeida, Vinicius Cardoso Garcia, and Silvio Romero de Lemos Meira. An experimental study to evaluate a {SPL} architecture regression testing approach. *{IEEE} 13th International Conference on Information Reuse {&} Integration, {IRI} 2012, Las Vegas, NV, USA, August 8-10, 2012*, pages 608–615, 2012.

[DGRN10] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, jul 2010.

[DKL09] Slawomir Duszynski, Jens Knodel, and Mikael Lindvall. {SAVE:} Software Architecture Visualization and Evaluation. In *13th European Conference on Software Maintenance and Reengineering, {CSMR} 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*, pages 323–324, 2009.

[DKO+97] David Dikel, David Kane, Steve Ornburn, William Loftus, and Jim Wilson. Applying Software Product-Line Architecture. *{IEEE} Computer*, 30(8):49–55, 1997.

[DKvDP15] Nicolas Dintzner, Uirá Kulesza, Arie van Deursen, and Martin Pinzger. Evaluating Feature Change Impact on Multi-product Line Configurations Using Partial Information. In *Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, {ICSR} 2015, Miami, FL, USA, January 4-6, 2015. Proceedings*, pages 1–16, 2015.

[DKZH12] Tejinder Dhaliwal, Foutse Khomh, Ying Zou, and Ahmed E Hassan. Recovering commit dependencies for selective code integration in software product lines. In *28th {IEEE} International Conference on Software Maintenance, {ICSM} 2012, Trento, Italy, September 23-28, 2012*, pages 202–211, 2012.

[DLHE14] Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. Automatic and incremental product optimization for software product lines. *Proceeding of the 7th International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40, 2014.

[DLS05] Gan Deng, Gunther Lenz, and Douglas C Schmidt. Addressing Domain Evolution Challenges in Software Product Lines. In *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, pages 247–261, 2005.

[DNGR08] Deepak Dhungana, Thomas Neumayer, Paul Grünbacher, and Rick Rabiser. Supporting Evolution in Model-Based Product Line Engineering. In *Software Product Lines, 12th International Conference, {SPLC} 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 319–328, 2008.

[DPG14] Jessica Díaz, Jennifer Pérez, and Juan Garbajosa. Agile product-line architecting in practice: A case study in smart grids. *Information & Software Technology*, 56(7):727–748, 2014.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 25–34, 2013.

[DRC13] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. Language Features for Software Evolution and Aspect-Oriented Interfaces: An Exploratory Study. *T. Aspect-Oriented Software Development*, 10:148–183, 2013.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: A case study. *Journal of Systems and Software*, 74(2 SPEC. ISS.):173–194, 2005.

[DSB09] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Variability assessment in software product families. *Information and Software Technology*, 51(1):1487–1510, 2009.

[Duv07] *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley, 2007.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Software Eng.*, 28(12):1146–1170, 2002.

[ER10] Emelie Engström and Per Runeson. A Qualitative Survey of Regression Testing Practices. In *Proceedings of the 11th International Conference on Product-Focused Software Process Improvement*, PROFES'10, pages 3–16, Berlin, Heidelberg, 2010. Springer-Verlag.

[ER11] Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *Inf. Softw. Technol.*, 53(1):2–13, 2011.

[FA75] Martin Fishbein and Icek Ajzen. Reading, Mass Addison-Wesley Pub. Co., 1975.

[Fav97] Jean-Marie Favre. Understanding-in-the-large. In *5th International Workshop on Program Comprehension (WPC '97), May 28-30, 1997 - Dearborn, MI, USA*, pages 29–38, 1997.

[FCS+08] Eduardo Figueiredo, Nélio Cacho, Cláudio Sant'Anna, Mario Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Cutigi Ferrari, Safoora Shakil Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects:

an empirical study on design stability. *30th International Conference on Software Engineering {(ICSE} 2008), Leipzig, Germany, May 10-18, 2008*, pages 261–270, 2008.

[FGFd14] Gabriel Coutinho Sousa Ferreira, Felipe Nunes Gaia, Eduardo Figueiredo, and Marcelo de Almeida Maia. On the use of feature-oriented programming for evolving software product lines - {A} comparative study. *Sci. Comput. Program.*, 93:65–85, 2014.

[FLLE16] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Software Engineering 2016, Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. Februar 2016, Wien, Österreich*, pages 95–96, 2016.

[FSK+16] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. Ten years of product line engineering at danfoss: lessons learned and way ahead. In *Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16-23, 2016*, pages 252–261, 2016.

[FV03] D. Faust and Chris Verhoef. Software product line migration and deployment. *Softw., Pract. Exper.*, 33(10):933–955, 2003.

[GCC+03] Akash Garg, Matt Critchlow, Ping Chen, Christopher van der Westhuizen, and André van der Hoek. An Environment for Managing Evolving Product Line Architectures. In *19th International Conference on Software Maintenance {(ICSM} 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, page 358, 2003.

[GF11] Nadia Gámez and Lidia Fuentes. Software Product Line Evolution with Cardinality-Based Feature Models. In *Top Productivity through Software Reuse - 12th International Conference on Software Reuse, {ICSR} 2011, Pohang, South Korea, June 13-17, 2011. Proceedings*, pages 102–118, 2011.

[GF13] Nadia Gámez and Lidia Fuentes. Architectural evolution of FamiWare using cardinality-based feature models. *Information and Software Technology*, 55(3):563–580, 2013.

[GFFd14] Felipe Nunes Gaia, Gabriel Coutinho Sousa Ferreira, Eduardo Figueiredo, and Marcelo de Almeida Maia. A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Sci. Comput. Program.*, 96:230–253, 2014.

[gita] Github flow. https://githubflow.github.io/.

[Gitb] A successful git branching model. http://nvie.com/posts/a-successful-git-branching-model.

[GLA+09] Dharmalingam Ganesan, Mikael Lindvall, Christopher Ackermann, David McComas, and Maureen Bartholomew. Verifying architectural design rules of the flight software product line. In *Software Product Lines, 13th International Conference, {SPLC} 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, pages 161–170, 2009.

[GP06] Jilles Van Gurp and Christian Prehofer. Version management tools as a basis for integrating Product Derivation and Software Product Families. *SPLC*, 2006.

[GpKL14] Thomas Devine Katerina Goseva-popstajanova, Sandeep Krishnan, and Robyn R Lutz. Assessment and cross-product prediction of SPL quality: accounting for reuse across products , over multiple releases. *Automated Software Engineering*, 2014.

[Gru07] Boris Gruschko. Changes classification in {M2} models. In *Software Engineering 2007 - Beitr{ä}ge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, 27.-30.3.2007 in Hamburg*, pages 277–280, 2007.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual {ACM} {SIGPLAN} Conference on Object-Oriented Programming, Systems, Languages, and Applications, {OOPSLA} 2003, October 26-30, 2003, Anaheim, CA, {USA}*, pages 16–27, 2003.

[GSLC14] Susan P. Gregg, Rick Scharadin, Eric Legore, and Paul Clements. Lessons from AEGIS : Organizational and Governance Aspects of a Major Product Line in a Multi-Program Environment. In *Proceedings of the 18th International Systems and Software Product Line Conference, SPLC*, 2014.

[GVSZ14] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 384–387, 2014.

[GWTB12] Jianmei Guo, Yinglin Wang, Pablo Trinidad, and David Benavides. Expert Systems with Applications Consistency maintenance for evolving feature models. *Expert Systems With Applications*, 39(5):4987–4998, 2012.

[Hev07] Alan R. Hevner. The three cycle view of design science. *Scandinavian J. Inf. Systems*, 19(2):4, 2007.

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series, 2010.

[HFG+10] Wolfgang Heider, Roman Froschauer, Paul Grünbacher, Rick Rabiser, and Deepak Dhungana. Simulating evolution in model-based product line engineering. *Information and Software Technology*, 52(7):758–769, 2010.

[HGR10] Wolfgang Heider, Paul Grünbacher, and Rick Rabiser. Negotiation constellations in reactive product line evolution. In *Fourth International Workshop on Software Product Management, {IWSPM} 2010, Sydney, NSW, Australia, September 27, 2010*, pages 63–66, 2010.

[HH07] Scott A Hendrickson and André Van Der Hoek. Modeling Product Line Architectures through Change Sets and Relationships. In *29th International Conference on Software Engineering (ICSE)*, pages 189–198, 2007.

[HMO+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 139–148, 2008.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[Hol12] Hannes Holdschick. Challenges in the evolution of model-based software product lines in the automotive domain. In *4th International Workshop on Feature-Oriented Software Development, {FOSD} '12, Dresden, Germany - September 24 - 25, 2012*, pages 70–73, 2012.

[HPMFA+15] Ruben Heradio, Hector Perez-Morago, David Fernandez-Amoros, Francisco Javier Cabrerizo, and Enrique Herrera-Viedma. A Bibliometric Analysis of 20 Years of Research on Software Product Lines. *Information and Software Technology*, 2015.

[HR10] W. Heider and R. Rabiser. Tool Support for Evolution of Product Lines through Rapid Feedback from Application Engineering. *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 167–170, 2010.

[HRG12] Wolfgang Heider, Rick Rabiser, and Paul Grünbacher. Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. *International Journal on Software Tools for Technology Transfer*, 14(5):613–630, 2012.

[HRGL12] Wolfgang Heider, Rick Rabiser, Paul Grünbacher, and Daniela Lettner. Using regression testing to analyze the impact of changes to variability models on products. In *16th International Software Product Line Conference, {SPLC} '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, pages 196–205, 2012.

[HSB] Robert Hellebrand, Michael Schulze, and Martin Becker. A Branching Model for Variability-Affected Cyber-Physical Systems.

[HVLG12] Wolfgang Heider, Michael Vierhauser, Daniela Lettner, and Paul Grünbacher. A Case Study on the Evolution of a Component-based Product Line. In *Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture (WICSA/ECSA)*, pages 1–10. Ieee, 2012.

[HZS+16] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. *Preprocessor-based variability in open-source and industrial software systems: An empirical study*, volume 21. 2016.

[IKH14] Mari Inoki, Takayuki Kitagawa, and Shinichi Honiden. Application of requirements prioritization decision rules in software product line evolution. In *5th {IEEE} International Workshop on Requirements Prioritization and Communication, RePriCo 2014, Karlskrona, Sweden, August 26, 2014*, pages 1–10, 2014.

[IMY+16] Takahiro Iida, Masahiro Matsubara, Kentaro Yoshimura, Hideyuki Kojima, and Kimio Nishino. PLE for automotive braking system with management of impacts from equipment interactions. *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*, pages 232–241, 2016.

[JB09] Hans Peter Jepsen and Danilo Beuche. Running a software product line: standing still is going backwards. In *Software Product Lines, 13th International Conference, {SPLC} 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, pages 101–110, 2009.

[JBAC15] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th International Conference on Software Product Line, {SPLC} 2015, Nashville, TN, USA, July 20-24, 2015*, pages 61–70, 2015.

[Jen07] P Jensen. Experiences with product line development of multi-discipline analysis software at overwatch textron systems. In *11th International Software Product Line Conference, SPLC 2007*, pages 35–43, Overwatch Textron Systems, 2007.

[JF88] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[JP14] Paul Johannesson and Erik Perjons. *An Introduction to Design Science*. Springer, 2014.

[JRG+12] Markus Jahn, Rick Rabiser, Paul Grünbacher, Markus Löberbauer, Reinhard Wolfinger, and Hanspeter Mössenböck. Supporting Model Maintenance in Component-based Product Lines. In *2012 Joint Working {IEEE/IFIP} Conference on Software Architecture and European Conference on Software Architecture, {WICSA/ECSA} 2012, Helsinki, Finland, August 20-24, 2012*, pages 21–30, 2012.

[JT11] Stan Jarzabek and Ha Duy Trung. Flexible generators for software reuse and evolution. In *Proceedings of the 33rd International Conference on Software Engineering, {ICSE} 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 920–923, 2011.

[JZZZ08] Michael Jiang, Jing Zhang, Hong Zhao, and Yuanyuan Zhou. Maintaining software product lines - an industrial practice. In *24th {IEEE} International Conference on Software Maintenance {(ICSM} 2008), September 28 - October 4, 2008, Beijing, China*, pages 444–447, 2008.

[Kan90] Kyo C Kang. *Feature-oriented Domain Analysis (FODA): Feasibility Study ; Technical Report CMU/SEI-90-TR-21 - ESD-90-TR-222*. Software Engineering Inst., Carnegie Mellon Univ., 1990.

[KB12] Anil Kumar and Bernd Bruegge. Issue-based variability management. *Information and Software Technology*, 54(9):933–950, 2012.

[KB13] Anil Kumar and Bernd Brügge. A mixed-method approach for the empirical evaluation of the issue-based variability modeling. *Journal of Systems and Software*, 86(7):1831–1849, 2013.

[KC05] Chang Hwan Peter Kim and Krzysztof Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *Model Driven Architecture - Foundations and Applications, First European Conference, {ECMDA-FA} 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings*, pages 331–348, 2005.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing {S}ystematic {L}iterature {R}eviews in {S}oftware {E}ngineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[KC13] Charles Krueger and Paul Clements. Systems and Software Product Line Engineering. *Encyclopedia of Software Engineering*, 2013.

[KDB+15a] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M. German. Open source-style collaborative development practices in commercial projects using GitHub. *Proceedings - International Conference on Software Engineering*, 1:574–585, 2015.

[KDB+15b] Eirini Kalliamvakou, Daniela E. Damian, Kelly Blincoe, Leif Singer, and Daniel M. Germán. Open source-style collaborative development practices in commercial projects using github. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 574–585, 2015.

[KG09] Mahvish Khurum and Tony Gorschek. A systematic review of domain analysis solutions for product lines. *Journal of Systems and Software*, 82(12):1982–2003, 2009.

[KH12] Michael Kircher and Peter Hofman. Combining systematic reuse with Agile development. *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*, 1:215, 2012.

[KJK+06] Ronny Kolb, Isabel John, Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Experiences with product line development of embedded systems at testo AG. *Proceedings - 10th International Software Product Line Conference, SPLC 2006*, (01):172–181, 2006.

[Kla08] Holger Eichelberger Klaus Schmid. A Requirements-Based Taxonomy of Software Product Line Evolution. *Proceedings of the Third International ERCIM Symposium on Software Evolution*, 8, 2008.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242. 1997.

[KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR 2006), 22-24 March 2006, Bari, Italy*, pages 279–294, 2006.

[KR02] Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling, 2nd Edition*. Wiley, 2002.

[KR13] Reza Karimpour and Günther Ruhe. Bi-criteria genetic search for adding new features into an existing product line. In *1st International Workshop on Combining Modelling and Search-Based Software Engineering, CMSBSE at ICSE 2013, San Francisco, CA, USA, May 20, 2013*, pages 34–38, 2013.

[Kru01] Charles W Krueger. Easing the Transition to Software Mass Customization. In *Software Product-Family Engineering, 4th International Workshop, {PFE} 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*, pages 282–293, 2001.

[Kru03] Charles W. Krueger. Towards a taxonomy for software product lines. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, pages 323–331, 2003.

[Kru06] Charles W. Krueger. New methods in software product line practice. *Commun. ACM*, 49(12):37–40, 2006.

[KS94] Maren Krone and Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, pages 49–57, 1994.

[KSK08] George Kakarontzas, Ioannis Stamelos, and Panagiotis Katsaros. Product Line Variability with Elastic Components and Test-Driven Development. In *2008 International Conferences on Computational Intelligence for Modelling, Control and Automation {(CIMCA} 2008), Intelligent Agents, Web Technologies and Internet Commerce {(IAWTIC} 2008), Innovation in Software Engineering {(ISE} 2008), 10-12 December*, pages 146–151, 2008.

[KSL+13] Sandeep Krishnan, Chris Strasburg, Robyn R Lutz, Katerina Goseva-popstojanova, and Karin S Dorman. Predicting failure-proneness in an evolving software product line. *Information and Software Technology*, 55(8):1479–1495, 2013.

[KSLG11] Sandeep Krishnan, Chris Strasburg, Robyn R. Lutz, and Katerina Goseva-Popstojanova. Are change metrics good predictors for an evolving software product line? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering, PROMISE 2011, Banff, Alberta, Canada, September 20-21, 2011*, page 7, 2011.

[KSS15] Michael Käßmeyer, Michael Schulze, and Markus Schurius. A process to support a systematic change impact analysis of variability and safety in automotive functions. In *Proceedings of the 19th International Conference on Software Product Line, {SPLC} 2015, Nashville, TN, USA, July 20-24, 2015*, pages 235–244, 2015.

[KST+14] R Kodama, J Shimabukuro, Y Takagi, S Koizumi, and S Tano. Experiences with commonality control procedures to develop clinical instrument system. *18th International Software Product Line Conference, SPLC 2014*, 1:254–263, 2014.

[KTvM+99] Barbara A Kitchenham, Guilherme H Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. Towards an Ontology of Software Maintenance. *Journal of Software Maintenance*, 11(6):365–389, 1999.

[LBd+13] Luanna Lopes Lobato, Thiago Jabur Bittar, Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Risk Management in Software Product Line Engineering: a Mapping Study. *International Journal of Software Engineering and Knowledge Engineering*, 23(4):523–558, 2013.

[LC13] Miguel A Laguna and Yania Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Program.*, 78(8):1010–1034, 2013.

[LDSL07] Jing Liu, Josh Dehlinger, Hongyu Sun, and Robyn R Lutz. State-Based Modeling to Support the Evolution and Maintenance of Safety-Critical Software Product Lines. In *14th Annual {IEEE} International Conference and Workshop on Engineering of Computer Based Systems {(ECBS} 2007), 26-29 March 2007, Tucson, Arizona, {USA}*, pages 596–608, 2007.

[LG15] Daniela Lettner and Paul Grünbacher. Using Feature Feeds to Improve Developer Awareness in Software Ecosystem Evolution. *Proceedings of the 9th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '15)*, pages 11–18, 2015.

[Liv11] Steve Livengood. Issues in software product line evolution: complex changes in variability models. In *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering, {PLEASE} 2011, Waikiki, Honolulu, HI, USA, May 22-23, 2011*, pages 6–9, 2011.

[LLSG12] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based {SPL} regression testing. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering, {PLEASE} 2012, Zurich, Switzerland, June 4, 2012*, pages 53–56, 2012.

[LP07] Felix Loesch and Erhard Ploedereder. Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. In *11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems, {CSMR} 2007, 21-23 March 2007, Amsterdam, The Netherlands*, pages 159–170, 2007.

[LRZJ04] Neil Loughran, Awais Rashid, Weishan Zhang, and Stan Jarzabek. Supporting Product Line Evolution With Framed Aspects. In *Proceedings of the 3rd workshop on Aspects, components, and patterns for infrastructure software (ACP4IS)*, pages 22–26, 2004.

[MAI12] Sonia Montagud, Silvia Abrahão, and Emilio Insfrán. A systematic review of quality attributes and measures for software product lines. *Software Quality Journal*, 20(3-4):425–486, 2012.

[MARC13] Alexandr Murashkin, Michal Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *17th International Software Product Line Conference, {SPLC} 2013, Tokyo, Japan - August 26 - 30, 2013*, pages 111–115, 2013.

[MBKM08] Thilo Mende, Felix Beckwermert, Rainer Koschke, and Gerald Meier. Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In *12th European Conference on Software Maintenance and Reengineering, {CSMR} 2008, April 1-4, 2008, Athens, Greece*, pages 163–172, 2008.

[Mcg03] John D Mcgregor. The Evolution of Product Line Assets. *Technical Report. Carnegie Mellon University, Software Engineering Institute*, 10(CMU/SEI-2003-TR-005), 2003.

[McG07] John McGregor. CM - configuration change management. *Journal of Object Technology*, 6(1):7–15, 2007.

[MCNY07] Mikyeong Moon, Heung Seok Chae, Taewoo Nam, and Keunhyuk Yeom. A metamodeling approach to tracing variability between requirements and architecture in software product lines. In *Seventh International Conference on Computer and Information Technology (CIT 2007), October 16-19, 2007, University of Aizu, Fukushima, Japan*, pages 927–933, 2007.

[McV15] Larry McVoy. Preliminary product line support in BitKeeper. In *Proceedings of the 19th International Conference on Software Product Line, {SPLC} 2015, Nashville, TN, USA, July 20-24, 2015*, pages 245–252, 2015.

[MD15] Leticia Montalvillo and Oscar Díaz. Tuning github for SPL development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 111–120, 2015.

[MD16] Leticia Montalvillo and Oscar Díaz. Requirement-driven Evolution in Software Product Lines: A Systematic Mapping Study. *The Journal of Systems and Software*, 2016.

[MDA17] Leticia Montalvillo, Oscar Díaz, and Maider Azanza. Visualizing product customization efforts for spotting SPL reuse opportunities. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017*, pages 73–80, 2017.

[mer] Ward Cunningham. Integration Hell. `http://c2.com/cgi/wiki?IntegrationHell`. Accessed: 2018-03-26.

[MKR94] K.R.S. Murthy, Anantha Kadur, and Padma Rao. A holistic approach to product marketability measurements-the pmm approach. In *Engineering Management Conference, 1994. 'Management in Transition: Engineering a Changing World', Proceedings of the 1994 IEEE International*, pages 323–329, 1994.

[MMCdA14] Ivan Do Carmo Machado, John D McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, oct 2014.

[MNSD17] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 467–478, 2017.

[MPC01] Kieran Mathieson, Eileen Peacock, and Wynne W. Chin. Extending the technology acceptance model: the influence of perceived user resources. *DATA BASE*, 32(3):86–112, 2001.

[MPK12] Daniel Merschen, Julian Pott, and Stefan Kowalewski. Integration and Analysis of Design Artefacts in Embedded Software Development. In *36th Annual {IEEE} Computer Software and Applications Conference Workshops, {COMPSAC} 2012, Izmir, Turkey, July 16-20, 2012*, pages 503–508, 2012.

[MPT07] Jose-Norberto Mazón, Jesús Pardillo, and Juan Trujillo. A model-driven goal-oriented requirement engineering approach for data warehouses. In *Advances in Conceptual Modeling - Foundations and Applications, ER 2007 Workshops CMLSA, FP-UML, ONISW, QoIS, RIGiM,SeCoGIS, Auckland, New Zealand, November 5-9, 2007, Proceedings*, pages 255–264, 2007.

[MV09] Radoslav Menkyna and Valentino Vranic. Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling. In *Advances in Software Engineering Techniques - 4th {IFIP} {TC} 2 Central and East European Conference on Software Engineering Techniques, {CEE-SET} 2009, Krakow,*

Poland, October 12-14, 2009. Revised Selected Papers, pages 40–53, 2009.

[MW11] Bartosz Michalik and Danny Weyns. Towards a Solution for Change Impact Analysis of Software Product Line Products. In *9th Working {IEEE/IFIP} Conference on Software Architecture, {WICSA} 2011, Boulder, Colorado, USA, June 20-24, 2011*, pages 290–293, 2011.

[MWB11] Bartosz Michalik, Danny Weyns, and Wim Van Betsbrugge. On the problems with evolving Egemin's software product line. In *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering, {PLEASE} 2011, Waikiki, Honolulu, HI, USA, May 22-23, 2011*, pages 15–19, 2011.

[MYBM91] Allan MacLean, Richard M Young, Victoria M E Bellotti, and Thomas P Moran. Questions, Options, and Criteria: Elements of Design Space Analysis. *Hum.-Comput. Interact.*, 6(3):201–250, 1991.

[NNK16] Motoi Nagamine, Tsuyoshi Nakajima, and Noriyoshi Kuno. A case study of applying software product line engineering to the air conditioner domain. *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*, pages 220–226, 2016.

[NRG08] Muhammad Asim Noor, Rick Rabiser, and Paul Grünbacher. Agile product line planning: A collaborative approach and a case study. *Journal of Systems and Software*, 81(6):868–882, 2008.

[PBD+12] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 85(10):2261–2274, oct 2012.

[PBvdL05a] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[PBvdL05b] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PCF14] J A Pereira, K Constantino, and E Figueiredo. A systematic literature review of software product line management tools. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8919:73–89, 2014.

[PDŠ12] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. Change impact analysis of feature models. *Communications in Computer and Information Science*, 319 CCIS:108–122, 2012.

[PFMM08a] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pages 68–77, Swinton, UK, UK, 2008. British Computer Society.

[PFMM08b] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. *12th International Conference on Evaluation and Assessment in Software Engineering, {EASE} 2008, University of Bari, Italy, 26-27 June 2008*, 2008.

[PGT+13] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. Coevolution of variability models and related artifacts: A case study from the Linux kernel. *ACM International Conference Proceeding Series*, pages 91–100, 2013.

[PHS11] Christian Pichler, Christian Huemer, and Michael Strommer. Evolution patterns for business document models. In *Software Product Lines - 15th International Conference, {SPLC} 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*, page 21, 2011.

[Pla] {Planning Game} Agile Practice. `http://c2.com/cgi/wiki?PlanningGame.Lastvisited:2015-12-11`. Accessed: 2018-02-23.

[PO97] T Troy Pearse and Paul W Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *ICSM*, pages 270–277, 1997.

[PPF+14] Juliana Padilha, Juliana Alves Pereira, Eduardo Figueiredo, Jussara M Almeida, Alessandro Garcia, and Cláudio Sant'Anna. On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study. In *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 656–671, 2014.

[Pre97] Christian Prehofer. Feature-Oriented Programming: {A} Fresh Look at Objects. In *ECOOP*, pages 419–443, 1997.

[PSW11] Shaun Phillips, Jonathan Sillito, and Rob Walker. Branching and Merging: An Investigation into Current Version Control Practices. *Proceedings of the 4th International Workshop on Cooperative and*

Human Aspects of Software Engineering SE - CHASE '11, pages 9–15, 2011.

[PTS+16] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing software variants with variantsync. *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*, pages 329–332, 2016.

[pur] Pure::variants. variant management tool from pure-systems company. `http://www.pure-systems.com/products/pure-variants-9.html`. Accessed: 2018-02-23.

[PVK15] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[PW09] Kai Petersen and Claes Wohlin. Context in industrial software engineering research. *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, pages 401–404, 2009.

[PYZ11] Xin Peng, Yijun Yu, and Wenyun Zhao. Analyzing Evolution of Variability in a Software Product Line: From Contexts and Requirements to Features. *Information and Software Technology*, 53(7):707–721, 2011.

[QPB+14] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. Consistency checking for the evolution of cardinality-based feature models. *Proceedings of the 18th International Software Product Line Conference (SPLC)*, pages 122–131, 2014.

[RB08] Márcio Ribeiro and Paulo Borba. Recommending Refactorings when Restructuring Variabilities in Software Product Lines. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 8:1—-8:4, New York, NY, USA, 2008. ACM.

[RBK14] Márcio Ribeiro, Paulo Borba, and Christian Kästner. Feature maintenance with emergent interfaces. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 989–1000, 2014.

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing Cloned Variants: A Framework and Experience. *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 101–110, 2013.

[RCC15] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Cloned product variants: from ad-hoc to managed software product lines. *STTT*, 17(5):627–646, 2015.

[RDG+07] R Rabiser, D Dhungana, P Grunbacher, K Lehner, and C Federspiel. Involving Non-Technicians in Product Derivation and Requirements Engineering: A Tool Suite for Product Line Engineering. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 367–368, oct 2007.

[RKBC12] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing forked product variants. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, pages 156–160, 2012.

[RR03] Claudio Riva and Christian Del Rosso. Experiences with software product family evolution. pages 161–169, 2003.

[RRSW] Bernhard Rumpe, Jan Oliver Ringert, Christoph Schulze, and Michael Von Wenckstern. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. *Proceedings of the 19th International Conference on Software Product Line (SPLC)*, pages 141–150.

[RUQ+13] Daniel Romero, Simon Urli, Clément Quinton, Mireille Blay-Fornarino, Philippe Collet, Laurence Duchien, and Sébastien Mosser. SPLEMMA: A Generic Framework for Controlled-Evolution of Software Product Lines. *Proceedings of the 17th International Software Product Line Conference co-located workshops (SPLC)*, 2013:59, 2013.

[Sav14] Juha Savolainen. Past, present and future of product line engineering in industry: reflecting on 15 years of variability management in real projects. In *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*, page 1:1, 2014.

[SB99] Mikael Svahnberg and Jan Bosch. Evolution in Software Product Lines: Two Cases. *Journal of Software Maintenance*, 11(6):391–422, 1999.

[SB00] Mikael Svahnberg and Jan Bosch. Issues Concerning Variability in Software Product Lines. *International Workshop on Software Architectures for Product Families (IW-SAPF).*, pages 146–157, 2000.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, pages 77–91, 2010.

[SC11] Ricardo J Sales and Roberta Coelho. Preserving the exception handling design rules in software product line context: A practical approach. In *Proceedings of the 5th Latin-American Symposium on Dependable Computing Workshops (LADCW)*, pages 9–16, 2011.

[Sch06a] H Schackmann H.; Lichter. A Cost-Based Approach to Software Product Line Management. *International Workshop on Software Product Management (IWSPM)*, pages 2–7, 2006.

[Sch06b] Kathrin D Scheidemann. Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems. *1Proceedings of the 10th International Software Product Line Conference (SPLC)*, 2006.

[SdOdA15] Alcemir Rodrigues Santos, Raphael Pereira de Oliveira, and Eduardo Santana de Almeida. Strategies for Consistency Checking on Software Product Lines: A Mapping Study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 5:1—-5:14, New York, NY, USA, 2015. ACM.

[SEB] Sebok. maintainability. `http://sebokwiki.org/wiki/Reliability,{_}Availability,{_}and{_}Maintainability`. Accessed: 2018-02-23.

[sem] Martin Fowler. Semantic conflict. `https://martinfowler.com/bliki/SemanticConflict.html/`. Accessed: 2018-03-26.

[Sha99] David C Sharp. Exploiting object technology to support product variability. *Proceedings of the 18th Digital Avionics Systems Conference*, 2, 1999.

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Assmann. Co-evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, pages 76–85, 2012.

[Sin98] Janice Singer. Practices of software maintenance. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 139–145, 1998.

[SK01] Juha Savolainen and Juha Kuusela. Violatility analysis framework for product lines. *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, 26:133–141, 2001.

[SK08] Juha Savolainen and Juha Kuusela. Scheduling Product Line Features for Effective Roadmapping. *Proceedings of the 5th Asia-Pacific Software Engineering Conference (APSEC)*, pages 195–202, 2008.

[SK14] Hamideh Sabouri and Ramtin Khosravi. Science of Computer Programming Reducing the verification cost of evolving product families using static analysis techniques. *Science of Computer Programming*, 83:35–55, 2014.

[SLB13] Sandro Schulze, Malte Lochau, and Saskia Brunswig. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, pages 33–40, New York, NY, USA, 2013. ACM.

[S.P] S.P.L.O.T. http://www.splot-research.org.

[SPP+13] Mathias Schubanz, Andreas Pleuss, Ligaj Pradhan, Goetz Botterweck, and Anil Kumar Thurimella. Model-driven planning and monitoring of long-term software product line evolution. *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, page 1, 2013.

[SPZ09] Liwei Shen, Xin Peng, and Wenyun Zhao. A Comprehensive Feature-Oriented Traceability Model for Software Product Line Development. *Australian Software Engineering Conference (ASWEC)*, pages 210–219, 2009.

[SPZZ10] Liwei Shen, Xin Peng, Jiayi Zhu, and Wenyun Zhao. Synchronized Architecture Evolution in Software Product Line Using Bidirectional Transformation. *Proceedings of the 34th Annual Computer Software and Applications Conference*, pages 389–394, 2010.

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 119–126, New York, NY, USA, 2011. ACM.

[SS08] Nita Sarang and Mukund A Sanglikar. *An Analysis of Effort Variance in Software Maintenance Projects*. 2008.

[SSRS16] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. Aligning Coevolving Artifacts Between Software Product Lines and Products. *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16*, pages 9–16, 2016.

[SSTS14] Reimar Schröter, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Feature-context interfaces: tailored programming interfaces for software product lines. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*, pages 102–111, 2014.

[Sta04] Mark Staples. Change control for product line software engineering. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea*, pages 572–573, 2004.

[STKS12] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. Variant-preserving Refactoring in Feature-oriented Software Product Lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 73–81, New York, NY, USA, 2012. ACM.

[SV02] K Schmid and M Verlage. The economic impact of product line adoption and evolution. *IEEE Software*, 19(4), 2002.

[Swa76] E Burton Swanson. The Dimensions of Maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Tab04] Louis J M Taborda. Generalized Release Planning for Product Line Architectures. *Proceedings of the 3rd International Conference on Software Product Lines (SPLC)*, pages 238–254, 2004.

[TABG15] Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. A product line of theories for reasoning about safe evolution of product lines. In *Proceedings of the 19th International Conference on Software Product Line, {SPLC} 2015, Nashville, TN, USA, July 20-24, 2015*, pages 161–170, 2015.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014.

[TB07] Anil Kumar Thurimella and Bernd Bruegge. Evolution in product line requirements engineering: A rationale management approach. *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE)*, pages 254–257, 2007.

[TBC08] Anil Kumar Thurimella, Bernd Bruegge, and Oliver Creighton. Identifying and exploiting the similarities between rationale management and variability management. *Proceedings of the 12th International Software Product Line Conference (SPLC)*, pages 99–108, 2008.

[TBG15] Leopoldo Teixeira, Paulo Borba, and Rohit Gheyi. Safe evolution of product populations and multi product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 171–175, 2015.

[TBK09] Thomas Thüm, Don Batory, and Christian Kastner. Reasoning about edits to feature models. *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 254–264, 2009.

[TBM+12] Christian Tischer, Birgit Boss, Andreas Müller, Andreas Thums, Klaus Schmid, Christian Tischer, Birgit Boss, Andreas Mueller, and Andreas Thums. Developing Long-Term Stable Product Line Architectures. *Proceedings of the 16th International Software Product Line Conference (SPLC)*, pages 86–95, 2012.

[tBMP11] Maurice H ter Beek, Henry Muccini, and Patrizio Pelliccione. Guaranteeing Correct Evolution of Software Product Lines: Setting Up the Problem. In *Software Engineering for Resilient Systems - Third International Workshop, {SERENE} 2011, Geneva, Switzerland, September 29-30, 2011. Proceedings*, pages 100–105, 2011.

[tBMP12] Maurice H ter Beek, Henry Muccini, and Patrizio Pelliccione. Assume-Guarantee Testing of Evolving Software Product Line Architectures. In *Software Engineering for Resilient Systems - 4th International Workshop, {SERENE} 2012, Pisa, Italy, September 27-28, 2012. Proceedings*, pages 91–105, 2012.

[TDR+11] Leonardo P Tizzei, Marcelo Dias, Cecília M F Rubira, Alessandro Garcia, and Jaejoon Lee. Components meet aspects : Assessing design stability of a software product line. *Information and Software Technology*, 53(2):121–136, 2011.

[Tes07] Aleksandra Tesanovic. Evolving embedded product lines: opportunities for aspects. In *Proceedings of the 6th workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS 2007, Vancouver, British Columbia, Canada, March 12, 2007*, page 10, 2007.

[TFC+09] Yasuaki Takebe, Naohiko Fukaya, Masaki Chikahisa, Toshihide Hanawa, and Osamu Shirai. Experiences with software product line engineering in product development oriented organization. *Proceedings of the 13th International Software Product Line Conference*, pages 275–283, 2009.

[TGAS14] Dan Tofan, Matthias Galster, Paris Avgeriou, and Wes Schuitema. Past and future of software architectural decisions ? A systematic mapping study. *Information and Software Technology*, 56(8):850–872, 2014.

[tM10] Adriaan ter Mors. *The world according to MARP*. PhD thesis, Delft University of Technology, Netherlands, 2010.

[TM14] Le Minh Sang Tran and Fabio Massacci. An Approach for Decision Support on the Uncertainty in Feature Model Evolution. In *{IEEE} 22nd International Requirements Engineering Conference, {RE} 2014, Karlskrona, Sweden, August 25-29, 2014*, pages 93–102, 2014.

[TMMK11] C Tischer, A Müller, T Mandl, and R Krause. Experiences from a large scale software product line merger in the automotive domain. In *15th International Software Product Line Conference, SPLC 2011*, pages 267–276, Robert Bosch GmbH, Diesel Gasoline Systems, Postfach 30 02 20, 70442 Stuttgart, Germany, 2011.

[TMN08] Cheng Thao Cheng Thao, E.V. Munson, and T.N. Nguyen. Software Configuration Management for Product Derivation in Software Product Families. *ECBS*, 2008.

[tru] What is your branching model. http://paulhammant.com/2013/12/04/what_is_your_branching_model/.

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 81–86, New York, NY, USA, 2009. ACM.

[uni] Unified diff format explained by Guido van van Rossum. `http://www.artima.com/weblogs/viewpost.jsp?thread=164293`. Accessed: 2018-02-23.

[VDJ10] Karina Villela, Jörg Dörr, and Isabel John. Evaluation of a method for proactively managing the evolving scope of a software product line. *International Working Conference on Requirements Engineering. Foundation for Software Quality (REFSQ)*, pages 113–127, 2010.

[vdLSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007.

[VFAC14] G Vale, E Figueiredo, R Abilio, and H Costa. Bad Smells in Software Product Lines: A Systematic Review. In *Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium on*, pages 84–94, sep 2014.

[vGB02] Jilles van Gurp and Jan Bosch. Design Erosion: Problems and Causes. *J. Syst. Softw.*, 61(2):105–119, 2002.

[VGH+12] Michael Vierhauser, Paul Grünbacher, Wolfgang Heider, Gerald Holl, and Daniela Lettner. Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines. In *Proceedings of the 15th International Conference*

on Model Driven Engineering Languages and Systems (MoDELS), pages 531–545, 2012.

[vO02] Rob van Ommering. Building product populations with software components. In *International Conference on Software Engineering (ICSE).*, 2002.

[VPS+12] Alexandre Vianna, Felipe Pinto, Demóstenes Sena, Uirá Kulezsa, Roberta Coelho, Jadson Santos, Jalerson Lima, and Gleydson Lima. Squid: An Extensible Infrastructure for Analyzing Software Product Line Implementations. In *Proceedings of the 16th International Software Product Line Conference (SPLC)- Volume 2*, volume II, pages 209–216, 2012.

[VRG14] Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. A Requirements Monitoring Infrastructure for Very-Large-Scale Software Systems. In Camille Salinesi and Inge van de Weerd, editors, *Requirements Engineering: Foundation for Software Quality*, volume 8396 of *Lecture Notes in Computer Science*, pages 88–94. Springer International Publishing, 2014.

[VV11] Markus Voelter and Eelco Visser. Product Line Engineering Using Domain-Specific Languages. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 70–79, Washington, DC, USA, 2011. IEEE Computer Society.

[WD15] Roel Wieringa and Maya Daneva. Six strategies for generalizing software engineering theories. *Science of Computer Programming*, 101:136 – 152, 2015. Towards general theories of software engineering.

[Wei08] David M Weiss. The Product Line Hall of Fame. In *Software Product Lines, 12th International Conference, {SPLC} 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, page 395, 2008.

[Wie14] Roel J. Wieringa. *Research Goals and Research Questions*, pages 13–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[WMHB11] Danny Weyns, Bartosz Michalik, Alexander Helleboogh, and Nelis Boucke. An Architectural Approach to Support Online Updates of Software Product Lines. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 204–213, Washington, DC, USA, 2011. IEEE Computer Society.

[WMMR05] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion. *Requir. Eng.*, 11(1):102–107, 2005.

[WRdMSN+13] Claes Wohlin, Per Runeson, Paulo Anselmo da Mota Silveira Neto, Emelie Engström, Ivan do Carmo Machado, and Eduardo Santana de Almeida. On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, 86(10):2594–2610, 2013.

[WS02a] Charlene (Chuck) Walrad and Darrel Strom. The importance of branching models in SCM. *IEEE Computer*, 35(9):31–38, 2002.

[WS02b] Chuck Walrad and Darrel Strom. The Importance of Branching Models in {SCM}. *Computer*, 35(9):31–38, 2002.

[YCM93] S S Yau, J S Collofello, and T M MacGregor. Software Engineering Metrics I. chapter Ripple Eff, pages 71–82. McGraw-Hill, Inc., New York, NY, USA, 1993.

[YGW12] Dongjin Yu, Peng Geng, and Wei Wu. Constructing Traceability between Features and Requirements for Software Product Line Engineering. *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC)*, pages 27–34, 2012.

[YM12] Amir Reza Yazdanshenas and Leon Moonen. Fine-Grained Change Impact Analysis for Component-Based Product Families. *Proceedings of the International Conference on Software Maintenance (ICSM)*, (5):119–128, 2012.

[ZBP+13] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability Evolution and Erosion in Industrial Product Lines: A Case Study. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 168–177, 2013.

Glossary

- **Application Engineering (AE)** is the process of developing a specific product for the needs of a particular customer (or other stakeholder). It corresponds to the process of single application development in traditional software engineering, but reuses artifacts from domain engineering where possible [ABKS13a].
- **Branching model.** A branching model embodies the rationales adopted for branching and merging configuration items within a Version Control Systems [WS02b]. It closely matches a team's software development process: it tells (1) how developers develop and collaborate with each other for new development, (2) how engineers release software both to test department and customers, and (3) how they deal with production fixes, i.e bugs that occur to the software released to customers. There is no a one-fit-all branching model, and each team needs to find its own.
- **Code peering** (or peering) refers to the practice that takes place during product development, whereby product engineers inspect and compare other products' code with their own code, and if interested, merge the other product's code into his/her own product. Code peering is intended to promote early reuse of product developments across product teams, with the aim of lessening the merge problem during pruning.
- **Core-assets** (or **core-asset base**) refer to domain engineering artifacts built "for reuse". These can source code, requirement documents, domain models, and test assets.
- **Core-asset release** (or **SPL release**) refers to the set of core-asset, tested and ready to be reused by application engineering teams.
- **Customization analysis** refers to the practice by which SPL engineers analyzing how products have changed the core-assets they were derived from. Customization analysis is intended to help SPL engineers identify interesting customizations to be promoted to reusable core-assets for the next core-asset release.
- **Design Science Research** is the scientific study and creation of artefacts as they are developed and used by people with the goal of solving practical problems of general interest.

- **Domain engineering (DE)** is the process of analyzing the domain of a product line and developing reusable artifacts (a.k.a core-assets). Domain engineering does not result in a specific software product, but prepares artifacts to be used in multiple, if not all, products of a product line [ABKS13a].

- **Feature.** A feature is prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [Kan90]. Features are used in product line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle [ABKS13a].

- **Feedback propagation** refers to the process of updating the core-asset base from product customizations that reside in already derived products.

- **Grow-and-prune model** refers to the approach of incrementally evolving an SPL from product customizations. During the *growth* seasons, products are allowed to customize the core-assets in order to attend to new customer needs, resolve bugs, or enhance functionalities. The *pruning* phase returns part of these product customizations to the core-asset base so they can be later reuse by products.

- **Merge problem** arises during the *pruning* stage of the *grow-and-prune* model, and refers to the issue of merging disparate product customizations, which result in a multitude of conflicts, and whose time to be resolved exceed the time it took to make the original changes.

- **Product derivation** (or product generation or product assembly) is the production step of application engineering, where reusable artifacts are combined according to the results of requirement analysis. Depending on the implementation approach, this process can be more or less automated, possibly, involving several development and customization tasks [ABKS13a]. This thesis considers product customization into product derivation.

- **Product customization** (or customization) takes place during product derivation, and refers to the process of changing the core-assets from which products were derived from, or create brand-new assets, in order to meet customer needs, or to resolve urgent bug-fixes.

- **Update propagation** refers to the process of updating already derived products with newer core-asset versions available in newer domain engineering.

- **Software Product Line Engineering (SPLE)** is the engineering of a portfolio of related products using a shared set of engineering assets and an efficient means of production [Kru06].

- **Software Product Line (SPLs).** A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core-assets in a prescribed way.

- **Version Control Systems (VCSs)** keep track of every change to a file over time so early versions can be restored and are used by software teams for source code.