Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Tag and Prune: A Pragmatic Approach to Software Product Line Implementation

Boucher, Quentin; Classen, Andreas; Heymans, Patrick; Bourdoux, Arnaud; Demonceau, Laurent

Published in:
25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), Antwerp, Belgium, 20-24 September 2010

Publication date:
2010

Document Version
Early version, also known as pre-print

Link to publication

Citation for pulished version (HARVARD):
Boucher, Q, Classen, A, Heymans, P, Bourdoux, A & Demonceau, L 2010, Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. in 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), Antwerp, Belgium, 20-24 September 2010. ACM Press, pp. 333-336.

Tag and Prune: A Pragmatic Approach to Software Product Line Implementation

Quentin Boucher, Andreas Classen,[*]
Patrick Heymans
PReCISE Research Centre
Faculty of Computer Science
University of Namur, Belgium
{qbo,acs,phe}@info.fundp.ac.be

Arnaud Bourdoux, Laurent Demonceau
Spacebel s.a.
Liège Science Park
{arnaud.bourdoux,laurent.demonceau}
@spacebel.be

ABSTRACT

To realise variability at the code level, product line methods classically advocate usage of inheritance, components, frameworks, aspects or generative techniques. However, these might require unaffordable paradigm shifts for the developers if the software was not thought at the outset as a product line. Furthermore, these techniques can be conflicting with a company's coding practices or external regulations.

These concerns were the motivation for the industry-university collaboration described in this paper where we develop a minimally intrusive coding technique based on tags. It is supported by a toolchain and is now in use in the partner company for the development of flight grade satellite communication software libraries.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Documentation, Languages

Keywords

Feature diagram, code tagging

1. INTRODUCTION

Software product line engineering (SPLE) is an increasingly popular software engineering paradigm institutionalising reuse across the software lifecycle. Central to the SPLE paradigm is the modelling and management of *variability*, i.e. *"the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts"* [18]. This variability is often conveniently expressed in terms of features, which appear to be high level abstractions that shape the reasoning of the engineers and other stakeholders [6]. A product of the SPL is seen as a set

---

[*]FNRS Research Fellow

of features. By adopting SPLE, one expects to achieve *mass-customisation*, i.e. the ability to create many different systems, leveraging on similarities between them and thereby improve the cost, productivity, time to market, quality of developing software.

Current approaches to the implementation of SPLs classically advocate usage of specific programming techniques (e.g., via inheritance, aspects,... [1, 2, 3, 4, 9, 19]), particular architectures (e.g., components or dedicated frameworks [9, 13]), compiler directives (e.g., `ifdef` directives in C [9, 17]), adapted IDEs (e.g., syntax colouring [8, 11, 15]) or generative programming [7]. One thing almost all of these approaches have in common is the paradigm shift they require in the way software is written. While the transition from 'classical' software engineering to SPLE is generally a conscious decision, known to have a major impact on project management, drastic changes to the development approach might prove unaffordable. In the case of safety-critical software, as for instance flight-grade satellite software, it can be outright impossible to change the coding paradigm, since it is part of the mission requirements and often enforced by external regulations (see, e.g., [16]).

These concerns were the main motivations for a collaboration between the University of Namur and an industrial partner, Spacebel S.A., a software company specialised in aerospace applications. The context of this work, which we now describe in more detail, is the development of a family of file transfer protocol libraries.

1.1 Industrial context

The CCSDS File delivery Protocol (CFDP) [5], is a file transfer protocol for communication links spanning interplanetary distances. The protocol was issued by the Con-
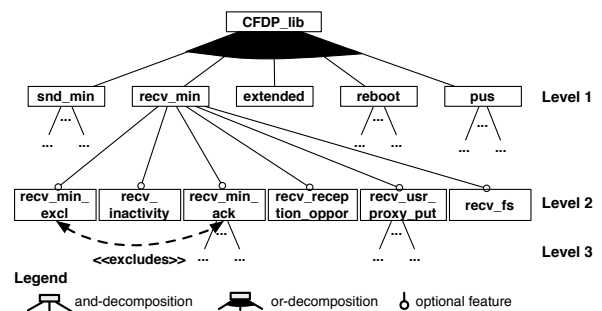

Figure 1: Partial FD of the CFDP library

sultative Committee for Space Data Systems (CCSDS). It is independent from the underlying file system and is meant to cover an extensive number of mission needs. Capabilities include deferred transmission, concurrent transfer, suspend/resume, and the ability to transmit via a proxy.

A partial feature diagram for the CFDP library is shown in Figure 1. Basically, *feature diagrams* (FDs) are trees whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. The features at level 1 represent the main functionality of the library: send (*snd_min*) and receive (*recv_min*) files, allow for a device to receive data through others (*extended*), safe reboot after unexpected system failure (*reboot*), and support for the "packet utilisation standard" (*pus*). Each is then further decomposed on one or two more levels, as shown for the *recv_min* feature.

Components for space usage are developed in order to deal with extreme environmental conditions such as cosmic rays, temperature variation and vibration. This type of hardware is thus generally very expensive and several evolutionary steps behind consumer hardware. As a consequence, very stringent restrictions apply to suppliers of on-board software components. CPU usage and memory footprint are typical quantities that have to be minimised. For such developments, practitioners often have no other choice than programming in *C* and obeying strict rules that prohibit usage of 'dangerous' mechanisms (typically, dynamic heap memory allocation) or general-purpose third-party libraries.

Considering these restrictions, and given that specific missions only require part of the protocol's functionality, it is highly desirable to only deploy those parts of the protocol that are eventually going to be used. More concretely, the feature set of a CFDP implementation has to be *minimal* wrt. the mission requirements: the implementation cannot include dead (i.e. *unused*) code. The SPLE approach described in this paper takes these constraints into account.

1.2 Problem statement and contribution

The objective of the collaboration between the university and Spacebel was to jointly develop an SPL implementation technique that would satisfy the following requirements:

(R1) Allow *mass-customisation* of the CFDP library, i.e. be able to efficiently derive products that only contain features required for a specific mission, such that *no product has dead code*.

(R2) Have a *minimal impact on current development practices*, and be *compliant with quality standards and regulations* in place for flight software.

(R3) *Automate* the solution as much as possible.

For the implementation we chose a pruning-based approach. At the code level, this is achieved by special annotations, called *feature tags*, which trace code fragments back to features (see Section 2). The overall process that spans from feature modelling down to compilation is described in Section 3. The approach is evaluated in Section 4.

2. TAG AND PRUNE

One can distinguish between two types of approaches for implementing SPLs: *compositional* approaches implement features as distinct modules while *annotative* approaches assume that there is one 'maximal' product where annotations

Figure 2: Code tagging example

in the source code indicate the feature a fragment belongs to [14, 15]. With the compositional approach, a product is generated by composing a set of fragments. With an annotative approach, a product is obtained by removing fragments corresponding to discarded features.

We decided to follow an annotative approach since in our case compositional approaches come with a paradigm shift, which would violate requirement R2. Furthermore, existing annotative approaches proved to be unsuitable for our undertaking.

Basically, our approach consists in annotating blocks of *C* code with *feature tags*; a feature tag being a list of the features that require the block to be present. If none of the features listed in a tag is included in a particular product, then the tagged code block will not be part of the source code generated for this product. Tags can also be nested and a whole file can be tagged with an additional annotation. Untagged code is assumed to be needed for all features.

Syntactically, a feature tag is a particular comment style. As such, it is displayed in the same colour as comments in code editors, which eases the reading. Our tags follow a predefined pattern that can be recognised by a feature parser.

```
<fcomment> ::= "/*@feature:" <flist> "@*/" [<filetag>]
<flist>    ::=   <featurename> ( ":" <flist> ) *
<filetag>  ::= "/*@!file_feature!@*/"
```

In this pattern, `<featurename>` identifies a feature of the FD.

The scope of a tag is the 'functional block', which we define as a group of statements that belong together, and that can be removed as a whole without violating the syntax or grammar of the language. Functional blocks thus correspond to elements of the abstract syntax tree (AST), an idea previously found in [15]. With this approach, we can guarantee that the pruned code will always be syntactically correct. Figure 2 is an example of tagged code where functional blocks associated to the different tags are highlighted.

Note that code tags were conceived as necessary conditions for code to be present, and so expressing the opposite of a given code tag (the 'else') can be tricky. However, this can be accomplished by defining additional features in the FD that exclude other features. The negation thus becomes implicit through the FD.

The main advantages of code tagging over the previously mentioned approaches are:

- the 'automatic' function block scoping: the developer does not need to track *closing* tags, or worry about syntactical correctness,

- its independence from a special IDE: the developer can use any code generator/editor combination.

To make sure that pruning always results in correctly typed code (a problem known as *safe composition* [21]), we had to set up particular coding guidelines and additional tests.

3. OVERALL PROCESS

The proposed process and its associated toolchain are represented graphically in Figure 3. It is organised after the classical SPLE process [18] which consists of two main streams: *domain engineering* (the creation of reusable artefacts) and *application engineering*. It uses 3 kinds of tools: (i) tools that were already in use, namely a UML modelling tool and a *C* compiler; (ii) tools built specifically to support the approach, namely the code pruner implementing the method from Section 2; and (iii) a commercial feature modelling tool.

Feature modelling. One of the first steps is to capture the variability of the SPL. In the case of the CFDP, we actually based this analysis on the official protocol specification [5]. The complete FD of the CFDP product line contains 75 features, four cross-tree constraints, and is up to three levels deep. The only supplemental expertise required from the engineer is knowledge about FDs, which turned out to be very intuitive to use. Feature modelling can be done with any FD tool. Tools with automated satisfiability checks and a configuration interface help automate some of the subsequent tasks.

Design. The main impact of our approach on the design phase is the need to take features into account when defining the architecture in order to facilitate implementation of the variability. In practice, this means that the architecture will tend to be more modular, so that high-level features directly map to high-level design artefacts, such as packages. We extended the UML tool in use at Spacebel so that engineers can tag design models with features, these tags being included in the code generated by the tool. The approach is general enough to accommodate any UML modelling tool that supports custom annotations. As in traditional development, a good design reduces development time; this effect

is amplified since the design has to account for the features that will eventually become tags in the code.

Implementation. In an effort to be minimally intrusive, implementation must remain largely untouched. The UML tool is used to generate code skeletons from the design models, including skeletons already tagged with features. The remaining code is written manually and tagged with features in the process. A few new coding rules were added due to the way the code pruner works, such as the mandatory use of blocks in *case* statements, for instance. The developer needs additional expertise: she has to understand the FD, and to learn the syntax and the semantics of the tagging language. The developers reported that the conceptual overhead caused by feature tags is manageable. During testing and debugging, around 20% of the errors were caused by feature tags, and only 5% of these were actual logical errors. The other 95% were all type errors, easily found and corrected. Feature tagging thus only caused a marginal increase in the number of errors.

Configuration. Configuration, that is, selecting the features to be included in a product, is done with the FD tool (reusing the FD elaborated during the feature modelling phase). The person doing the configuration needs to have a deep knowledge of the mission requirements as well as a sufficient understanding of the FD, to be able to map mission requirements to features of the library. It is during this activity that the initial investment pays off. If this activity were performed manually on untagged code, it would take around 20% of the development time to create each product. Furthermore, manual code pruning would be error-prone.

Code pruning and compilation. Up to this point, the implementation contains all possible features. The last step thus consists in removing non selected features from the tagged source code. To this end, we implemented a source code pruner for ANSI *C* enriched with the feature tag syntax described in Section 2. This parser takes as input the source files of an application and the list of features that make up the product to be created and creates pruned files as output. These can be used to compile the final product and thus constitute the end result of the tool-supported process.

4. EVALUATION

Here we discuss the extent to which the initial requirements, formulated in Section 1.2, were satisfied.

(R1) Mass-customisation and no dead code. The first requirement is the outset of the project: be able to quickly produce a reduced version of a library on demand. Mass-customisation is enabled by following an SPL approach, with explicit variability management through FDs. Dead code is avoided by pruning unnecessary code (related to features that are irrelevant for the specific needs of a mission) before compilation. We have conducted various experiments to measure the gains in memory and CPU footprint that can be achieved by customising a library to specific mission requirements: while the full library requires 65 Kb of PROM, a version restricted to sending files needs 16.2 Kb, four times less. To put this into perspective, for the LISA Pathfinder (a planned ESA mission which does not use the CFDP protocol), the PROM budget for the entire data handling system is 375 Kb. We therefore consider this requirement to be met.

(R2) Minimal impact and compliance. The application domain comes with a number of stringent development constraints, quality standards and regulations. As detailed
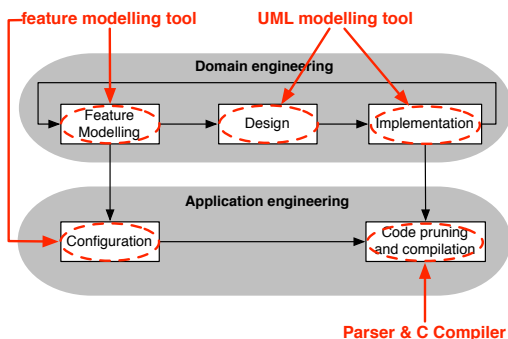

Figure 3: Toolchain as deployed at Spacebel

in the previous section, the transition to our SPLE approach necessitates three changes: (1) specification and design now include the FD, (2) code has to be developed with tags that trace back to features, and (3) each delivered library goes through a configuration and pruning step. These changes require additional expertise from engineers and developers: FDs and the tagging technique. According to the practitioners involved, both are very easy to master and do not affect coding practice in a fundamental way; the development environment and paradigm are not affected. Furthermore, none of these points impacts the technologies used during the development process. Technically, the deliverable is thus indistinguishable from one that would have been developed individually, which meets the compliance requirement.

(R3) Automation. All of the additional steps are tool-supported, and pruning is fully automated. There is room for improvement, though. Firstly, FD elaboration and configuration will always require user intervention and can thus never be completely automated. In previous papers, we proposed methods to further improve the configuration process based on the Spacebel case [12, 22]. Secondly, although the tagging approach keeps the code very readable, readability could be enhanced by highlighting tagged code fragments (e.g. with colours [15]). A first prototype of such a tool, based on TagSEA [20], has already been developed [10]. Thirdly, integration of the various tools in the toolchain of Figure 3 is currently not very tight in that it occurs only through file exchange.

5. CONCLUSION

To realise variation points at the code level, product line methods classically advocate usage of inheritance, components, frameworks, aspects or generative techniques. These techniques often require unaffordable paradigm shifts from the developers if the software was not thought at the outset as a product line. As part of a partnership between industry and university we developed a novel approach to implementing SPLs with three principal goals: (1) allow mass-customisation, (2) have a minimal impact on development practices, and (3) be automated. The approach uses FDs to capture variability and its kernel consists of a technique for tagging portions of code with features. It has been used successfully for the development of a flight grade satellite file transfer library product line.

Acknowledgements

6. REFERENCES

[1] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR*, pages 141–156, 2004.

[2] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34:162–180, 2008.

[3] D. Batory. Feature-oriented programming and the ahead tool suite. In *ICSE'04*, pages 702–703, 2004.

[4] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE 2003*, pages 187–197, 2003.

[5] CCSDS. *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4 and Green Book, Issue 3*. Number CCSDS 727.0-B-4, CCSDS 720.1-G-3. NASA, 2007.

[6] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In *FASE'08*, volume 4961 of *LNCS*, pages 16–30, 2008.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, 2000.

[8] P. Ebraert, A. Classen, P. Heymans, and T. D'Hondt. Feature diagrams for change-oriented programming. In *ICFI/FIW'09*. IOS Press, June 2009.

[9] C. Gacek and M. Anastasopoulos. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3):109–117, 2001.

[10] C. Gauthier, A. Classen, Q. Boucher, P. Heymans, M.-A. Storey, and M. Mendonca. XToF: A tool for tag-based product line implementation. In *VaMoS'10*, pages 163–166. University of Duisburg-Essen, 2010.

[11] F. Heidenreich, J. Kopcsek, and C. Wende. Featuremapper: mapping features to models. In *ICSE'08*, pages 943–944, 2008.

[12] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *SPLC'09*, pages 221–230, 2009.

[13] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. Xvcl: Xml-based variant configuration language. In *ICSE'03*, pages 810–811. IEEE CS, 2003.

[14] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *GPCE'08*, pages 35–40, 2008.

[15] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE'08*, 2008.

[16] MISRA. *MISRA-C: Guidelines for the use of the C language in critical systems*. Motor Industry Research Association, UK, 2008.

[17] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11):1, 2006.

[18] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Secaucus, NJ, USA, 2005.

[19] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[20] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *CSCW'06*, pages 195–198. ACM, 2006.

[21] S. Thaker, D. S. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE'07*, pages 95–104, 2007.

[22] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: A systematic approach. In *SPLC'09*, pages 201–210, 2009.