Towards a Product Derivation Process Framework

Padraig O’Leary, Steffen Thiel, Goetz Botterweck, Ita Richardson
Lero, the Irish Software Engineering Research Centre, University of Limerick, Ireland
{padraig.oleary, steffen.thiel, goetz.botterweck, ita.richardson}@lero.ie

Abstract. Inefficient product derivation practices can greatly diminish the productivity gains expected from a software product line approach. As a foundation for systematic and efficient product derivation a better understanding of the underlying activities in industrial product line development is required. We have developed a process framework that comprises important tasks product line stakeholders have to perform during product derivation. The framework is based on both literature and industrial practice. In this paper we report on observations obtained in a case study with an automotive supplier, describe our results to date in developing a product derivation process framework and outline how our framework can provide a link to automated derivation approaches.

Keywords: Software Product Lines, Product Derivation, Process Engineering

1 Introduction

1.1 Software Product Lines

A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying 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 [1]. The SPL approach makes a distinction between domain engineering, where a common platform for an arbitrary number of products is designed and implemented, and application engineering, where a product is derived based on the platform components [2]. The separation into domain engineering and application engineering allows the development of software artefacts which are shared among the products within that domain. These shared artefacts become separate entities in their own right, subscribing to providing shared functionality across multiple products.

It is during application engineering that the individual products within a product line are constructed. The products are built using a number of shared software artefacts created during domain engineering. The process of creating these individual products using the platform artefacts is known as product derivation.
1.2 Product Derivation

Product Derivation is the process of constructing a product from a Software Product Lines (SPL) core assets [3]. An effective product derivation process can help to ensure that the benefits delivered through using these shared artefacts across the products within a product line is greater than the effort required to develop the shared assets. In fact, the underlying assumption in SPL that “the investments required for building the reusable assets during domain engineering are outweighed by the benefits of rapid derivation of individual products” [4] might not hold if inefficient derivation practices diminishes the expected gains.

A number of publications speak of the difficulties associated with product derivation. Hotz et al. [5] describe the process as “slow and error prone even if no new development is involved”. Griss [6] identifies the inherent complexity and the coordination required in the derivation process by stating that “…as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved”. Therefore, the derivation of individual products from shared software assets is still a time-consuming and expensive activity in many organisations [3].

Despite this, there has been little work dedicated to the overall product derivation process. Rabiser et al. [7] claim that “guidance and support are needed to increase efficiency and to deal with complexity of product derivation”. As Deelstra et al. [3] states there “is a lack of methodological support for application engineering and, consequently, organizations fail to exploit the full benefits of software product families.”

1.3 Contribution

This paper presents the results to date of our research on the development of a Product Derivation Process Framework (PDPF). The preparatory stage of this research was conducted as a review of existing SPL whitepapers, product derivation papers and software process improvement (SPI) practices. These initial results were further developed and assessed through a series of iterative workshops over a four month period. Evidence and feedback from SPL practitioners and researchers was collected from these organised workshops. Case study research was performed the observations from which helped to augment and refine our framework.

The remainder of this paper is organised as follows: In Section 2, we describe case study research we conducted into industrial product derivation practices. In Section 3, we report on the observations from this case study. In Section 4 we present our Product Derivation Process Framework (PDPF). In Section 5, we present how the framework can be used as a foundation for automated approaches. In section 6, we discuss related work and finally in section 7, we present our conclusion and future work.
2 Case Study Research

For the case study, we collected data on the product derivation practices of a major supplier of automotive systems. The systems produced consist of both hardware (such as processors, sensors, connectors, and housing) and software.

Prior to an on-site visit of the case study company, we had access to internal company documentation. These documents included information on product derivation practices within a particular business unit, organisational structure of the company’s teams and information on various derivation techniques applied within the company.

For the onsite visit to the company, we organised a two day workshop. During the workshop we presented our preliminary findings on the company’s derivation practices and used these initial findings to drive the workshop discussion. In total three researchers facilitated the running of the workshop.

We have two primary outputs from this part of the research. Firstly, we created and documented a process model of the case study company’s derivation practices. Secondly, we extended our initial PDPF by generalising and discussing the three major observations from the case study company.

3 Case Study Observations

3.1 Additional Development Disciplines

We noted that the organisational structure is broken into three broad disciplines, software, hardware and mechanics. Within each of these disciplines there are further sub-disciplines. The software discipline had basic software and algorithms teams. The hardware discipline has a microcontroller team and an ECU (Electronic Control Unit) team. The mechanics discipline has housing, mechanical quality and interfaces and plugs teams.

3.2 Additional Roles and Tasks

During the case study we observed an unexpected number of product derivation roles. For example, the software product team consists of architects, developers, integrators, testers, and customer specific component developers. These roles are replicated across the independent product sub-discipline teams and platform teams. Moreover, similar roles exist for hardware and mechanics.

These intricate role structures are reflected by corresponding communication and task structures. For instance, the allocation of requirements to responsible teams has to consider the various disciplines and sub-disciplines. This requires a finer granularity of requirements management tasks than originally envisaged in our framework. We can see this when the case study company starts a product-specific
project. During the early phases, the customer requirements are translated into a set of internal company documents. These documents are processed and augmented during various tasks where requirements are analysed for reuse potential and then assigned to responsible disciplines and sub disciplines.

Another consequence of this distributed development across both disciplines and platform and product teams is the raised importance of modularisation. Consequently, interface management is performed as an explicit task and encapsulation is a key design property for component development; a software component should ideally be independent of how a sensor, actuator or microcontroller works internally.

3.3 Platform-Product Synchronisation

Within the case study company, given the heavy dependencies across disciplines and the platform product divide, product development requires a high degree of coordination and communication. In Figure 1, we illustrate the observed platform product dependencies.

The product team is responsible for designing and implementing customer-specific components based on the customer requirements. The platform team receives the platform software requirements containing the required extensions to the existing platform in order to facilitate the new customer requirements. Both the customer-specific and platform development is occurring in parallel. The product team needs to interface correctly with the new platform release. Here, the product team can choose between two alternative development strategies. Option 1 is to go ahead and design and implement customer-specific components using the old platform release, which has not yet been updated, as a basis for development. Consequently, when the new
platform architecture is released, the product team has to check the compatibility of the developed components with the new architecture.

Option 2 for the product team is to wait for the updated platform release. This is suggested when potentially large compatibility issues are expected with the risk of wasted development effort.

A third hybrid option, not illustrated in figure 1, is for the product team to first negotiate a platform interface with the platform team before proceeding to develop in parallel against the platform team. Alternatively, the product team can make assumptions on expected interface changes, and work from these expectations. After the updated platform release the product team check the compatibility of the developed components with the new architecture.

These outside dependencies that the product team must handle are a reoccurring process pattern within product derivation. Similar dependencies can be seen during software integration with the hardware modules. Again, the software product team can choose one of the described strategies, for handling the hardware interfaces during parallel software and hardware development.

3.4 More Documentation

The case study company relies heavily on documentation to drive the product derivation process. Documentation is used to facilitate communication and synchronise development between the product and platform teams, between the different hardware, software and mechanical disciplines and also between the sub-disciplines. It is also used as a milestone to plot project progress and as a driver to trigger certain tasks within the project. Additionally, in certain domains evidence of due process is required by law, some documents satisfy this condition.

4 Product Derivation Process Framework

In this section, we describe the phases of our PDPF. Within these phases we have identified both domain-specific tasks and non domain-specific tasks. The domain-specific tasks are those that we identified through our case study research into product derivation activities in embedded systems development. The non-domain-specific tasks are those which are identified as fundamental product derivation activities.

4.1 Formalising the Framework

We are using the Eclipse Process Framework (EPF) [8] to model the product derivation process and create a more formal version of our PDPF. EPF allows the development, maintenance and deployment of process content. It also allows the development of situational method content. By enabling inbuilt process variability within EPF we can select, tailor or remove content from a process in order to strike the right balance for a particular situation. This has the potential for making the PDPF as applicable to a small software development team working on a mobile application
as it is for a large aerospace and defence contractor building a system of systems. For instance, in the case study company we saw that embedded software development is a cross discipline activity. In this context, discipline mapping where requirements are allocated to software, hardware or mechanical disciplines, would be a relevant task. This process flexibility allows us to model generic product derivation practices and domain-specific practices.

### 4.2 Inclusion of Additional Roles

During the case study we observed an unexpected number of product derivation roles. For example, the software product team consists of architects, developers, integrators, testers, and customer specific component developers. These roles are replicated across the independent product sub-discipline teams and platform teams. Moreover, similar roles exist for hardware and mechanics.

These intricate role structures are reflected by appropriate communication and task structures. For instance, the allocation of requirements to responsible teams has to consider the various disciplines and sub-disciplines. This requires a finer granularity of requirements management tasks than originally envisaged in our framework. Another consequence of this distributed development across both disciplines and platform and product teams is the raised importance of modularisation. Consequently, interface management is performed as an explicit task and encapsulation is a key design property for component development; a software component should ideally be independent of how a sensor, actuator or microcontroller works internally.

![Fig. 2. Overview of the PDPF](image)

### 4.3 Overview of the PDPF

The framework is structured into four main phases. Figure 2 provides an overview of the framework, showing the interactions between the main phases:
1. **Impact Analysis** is aimed at gathering product-specific requirements based on customer requirements and negotiation with the platform team.

2. **Reusability Analysis** purports to create a partial product configuration based on the product-specific requirements and by using the available core assets.

3. During **Component Development and Adaptation**, new components are developed (if required) and existing components are adapted to satisfy requirements which could not be satisfied by existing core assets.

4. Finally, **Product Integration and Validation** aims to integrate the core asset configuration and newly developed components. The integrated product is then validated by performing appropriate testing procedures.

We will now discuss each of these product derivation phases in more detail.

### 4.4 Impact Analysis

The goal of ‘Impact Analysis’ is to create the product-specific requirements based on customer requirements and negotiation with the platform team. In Figure 3, we show the ‘Impact Analysis’ tasks modelled in EPF.

In Section 3, we discussed the requirements management process of the case study company. We observed how the customer requirements are used to create a system requirements specification which is in turn broken into individual discipline requirements, allocated to sub-discipline teams and categorised as platform or product requirements. As a result, we included the need for a more sophisticated requirements management process particularly when dealing with large distributed teams.

The task ‘Rationalising the CRS’ translates the customer requirements from a customer-specific document into an internal company-specific document. The product team uses the ‘Glossary’ document which contains customer terms and their Product Line equivalent as a guide during this task. In ‘Coverage analysis’ the product team determines those customer requirements which can be satisfied through a configuration of the platform assets. The results of this task are used during the ‘Customer Negotiation’ task.

![Fig. 3. ‘Impact Analysis’ modelled in EPF](image)

The ‘Discipline Mapping’ task allocates requirements to the relevant disciplines; the requirement allocation is held in separate requirements documents, such as the
platform software requirements specification and the customer hardware requirements specification. Finally, the product-specific test cases are created.

4.5 Reusability Analysis

The goal of the ‘Reusability Analysis’ phase is to create a partial product configuration that makes maximum use of the platform artefacts and minimises the amount of product-specific development required.

According to Deelstra et al., the product team can use one of three approaches to create a base product configuration, i.e., configuration selection, assembly or a hybrid of the former two approaches [3]. We have modelled these alternative derivation approaches in EPF through the development of process patterns (see Figure 4). Process patterns are clusters of reusable activities that can be used as building blocks for the construction of delivery processes. The process pattern applied to a particular product derivation project depends on the technology available to the product team.

In the case study company, the derivation approach used during Reusability Analysis was assembly. The product team constructed the derivation product through the derivation of the product architecture from the overall platform architecture, the selection of components for reuse and finally the setting of parameters for each selected component.

At this stage, if the customer requirements can be completely satisfied by a configuration of the platform assets then the product team can began product validation. However in many cases, some customer requirements will fall outside the scope of the platform. Customer requirements which could not be satisfied through reuse of existing platform assets are satisfied through component development and adaptation.
4.6 Component Development and Adaptation

The goal of the ‘Component Development and Adaptation’ phase is to satisfy requirements which could not be satisfied through the reuse of existing platform assets (See Figure 5). The product team firstly identifies those requirements which were not satisfied by the partial product configuration before identifying what component development or adaptation is required.

![Component Development and Adaptation modelled in EPF](image)

**Fig. 5.** ‘Component Development and Adaptation’ modelled in EPF

The decision of whether the required component development or adaptation will result in product-specific code or adaptation of the product line (platform) is determined through a Change Control Board (CCB) in the task ‘Scoping of Development’. If the CCB decides that the component development should occur at
the platform level then the platform team has to adapt or develop new shared artefacts and release a new version of the platform. Based on the new platform, the product team must repeat the Reusability Analysis for the products under consideration.

If the development or adaptation is designated to be product-specific then it is the responsibility of the product development team to implement the required component changes at the product level. For product-specific development, the product team must synchronise with the platform. In Section 3.3 we described the case study synchronisation process pattern used by the product team to handle development dependencies. We have extended the PDPF to handle these development dependencies. The product team can decide on an implementation strategy based on their development needs. Figure 6 shows the two development strategies for product-specific development which we have modelled in EPF.

**Option one.** The product team waits for the new platform release and then proceeds to design, implement and test customer-specific components.

**Option two.** The product team bases new component development on the existing platform architecture. The product team first negotiates a platform interface with the platform team before proceeding to develop in parallel. Alternatively, the product team will make assumptions on interface changes, working off expected changes to the interface. If conflicts are detected when the new platform architecture is released, then the product team makes the alterations.

### 4.7 Product Integration and Validation

The main goal of the ‘Product Integration and Validation’ is to prepare the product for final delivery to the customer (see Figure 7). If required this involves, integrating the partial product configuration from the ‘Reusability Analysis’ phase and the newly developed or adapted components from the ‘Component Development and Adaptation’ phase. The product team integrates the developed or adapted components and the partial product configuration by writing sufficient “glue” code to interface with the components [9]. This includes implementing any required architectural changes to facilitate the developed or adapted components.

Integration Testing validates the platform assets for this particular configuration. The integration tests should reuse platform test artefacts. This also ensures that no new errors appear due to the integration of core assets with product-specific assets [10].

![Fig. 7. Product Integration and Validation](image)

After ‘Integration Testing’, System testing is performed. System Testing verifies if the product as a whole conforms to the product-specific requirements. System test artefacts such as the product-specific test cases are already derived from the product-specific requirements [10] in phase 1, ‘Impact Analysis’. 
If the product fails ‘Integration Testing’ or system testing then the current configuration may not provide the required functionality, or some of the selected components simply do not work together as expected. In this case, the product team should repeat ‘Reusability Analysis’ or ‘Components Development and Adaptation’ phases depending on the scope of the required changes.

5 The Framework as a Foundation for Automated Approaches

In this section we discuss the contributions the PDPF can provide in the context of automated software engineering. We have derived our PDPF from literature and industry practice (Figure 8 left). We abstracted and modelled the discovered process structures into our process framework. Figure 8 shows the role the PDPF can play in bridging the gap between current industrial practices and the adoption of automated approaches. The PDPF can act as a roadmap towards automation. As we move from left to right, the need for further formalisation and abstraction of product derivation activities is required; the use of EPF is a means of achieving this required formalisation and abstraction.

Fig. 8. Industrial Practice, PDPF and ASE

When focussing on a particular task within product derivation, it is possible to use automated approaches. The PDPF can serve as a foundation for these approaches, by providing the bigger context and describing overall process structures for product
derivation. Within this context, approaches that focus on the automation of particular derivation steps can be set against the bigger product derivation picture.

One example for such approaches (Figure 8 right) is the derivation of application-specific architectures from the product-line architecture, PLA. This derivation is based on the domain feature model and the application-specific feature configuration.

The PLA contains variability to cover the full range of products which can be created from the product line. In the Application Architecture, after the derivation process, this variability is gone since the feature configuration is fixed and all decisions whether components should be included or not, have been implemented.

In earlier work in this topic [11] we described how this process can be fully automated, when we assume that the PLA contains enough elements to cover all possible Application Architectures and the process of derivation is thereby simplified to the task of filtering the right elements from the PLA. The filtering is directly based on the feature configuration and the links between domain-feature model and PLA which describes “realized-by” relationships between features and architectural components. In our approach the process of architecture derivation is implemented as an model transformation in the Atlas Transformation Language (ATL) [12].

6 Related Work

To date, several approaches and tools that support or partly automate product derivation activities in SPL have been proposed. Asikainen et al. [13] provide a product configuration modelling language (PCML) and configuration tool (WeCoTin). PCML supports the creation of feature models for a software product line. WeCoTin is used to derive valid feature models for particular products of the product line.

The ConIPF Methodology [5] proposed by Hotz et al. tackles the challenges of product derivation by combining concepts from product line engineering and knowledge-based configuration.

Rabiser et al. [7] present an approach for supporting product derivation using feature specifications. The approach introduces business decision-making into product derivation through a combination of modelling stakeholder needs, product features, architectural elements, and variability. The approach emphasises supporting the requirements acquisition and management mechanism through the use of variability models.

McGregor [14] introduces the production plan, which prescribes how products are produced from platform assets. It contains the attached processes of the platform assets as well as an overall scheme of how the processes are combined to build products. The product plan facilitates the passing of knowledge between the platform developers and the product developers. An example of the production plan in use is given in [9]. McGregor also provides an overview of technologies and approaches to automate product derivation.

Deelstra et al. [3] present a product derivation approach developed based on two industrial case studies. The framework consists of two phases: an initial and an
iteration phase. During the initial phase, a first product configuration is derived from the product line artefacts. The initial configuration is modified in a number of subsequent iterations during the iteration phase until the product sufficiently implements the imposed requirements. Requirements that cannot be accommodated by existing assets are handled by product-specific adaptation or reactive evolution. Parts of the derivation framework have been implemented in a research tool called COVAMOF [15], a variability modelling framework which purports to solve the product derivation problems associated with dependencies.

The work by Deelstra et al. [4] presents a framework of terminology and concepts for product derivation. The framework focuses on product configuration and is a high level attempt at providing the methodological support that Deelstra et al. agree is required for product derivation.

7 Conclusion and Future Work

This research is motivated by the assumption that despite the adoption of SPL within industry, product derivation remains an expensive and error-prone activity. We have presented a product derivation framework which is based on an extensive literature review, discussions with SPL practitioners and researchers and observations from case study research. We discuss the contribution that this framework can make to the automation of product derivation.

We are planning additional case studies, within other domains. This will help to further generalise our framework while also identifying those activities which are domain-specific. Our goal in the near future is to provide a version of the PDPF which is completely described by EPF. This allows us to employ variability in the process framework. Consequently, the models can be adapted and customized for a particular industry and organization.

As a further aspect of the product derivation processes, we are interested in the integration of agile practices with plan-driven approaches. We believe that in many contexts the adoption of agile practices can improve the product derivation process. Our work to date in this area has identified a set of agile practices that have potential for integration into the product derivation process [16, 17]. We believe that our framework can provide a means of balancing between agility and formalism during the product derivation process.

8 Acknowledgements

This work is partially funded by IRCSET under grant no. RS/06/167 and by Science Foundation Ireland under grant no. 03/CE2/I303_1.
9 References


