Downloaded from orbit.dtu.dk on: Jan 27, 2023

DTU Library

Adoption of Software Product Line from Extreme Derivative Development Process

Nakanishi, Tsuneo; Jæger-Hansen, Claes Lund; Griepentrog, Hans-Werner

Publication date:
2012

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Nakanishi, T., Jæger-Hansen, C. L., & Griepentrog, H-W. (2012). *Adoption of Software Product Line from Extreme Derivative Development Process*. Paper presented at 3rd International Conference on Machine Control & Guidance, Stuttgart, Germany.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Evolutional Development of Controlling Software for Agricultural Vehicles and Robots

Tsuneo Nakanishi[1], Claes Jæger-Hansen[2], and Hans-Werner Griepentrog[2]
[1]Faculty of Information Science and Electrical Engineering, Kyushu University, Japan
[2]Institute of Agricultural Engineering, University of Hohenheim, Germany

Abstract
Agricultural vehicles and robots expand their controlling software in size and complexity for their increasing functions. Due to repeated, *ad hoc* addition and modification, software gets structurally corrupted and becomes low performing, resource consuming and unreliable. This paper presents an evolutional development process combining *Software Product Line* (*SPL*) and *eXtreme Derivation Development Process* (*XDDP*). While SPL is a promising paradigm for successful reuse of software artefacts, it requires understanding of the whole system, a global and future view of the system, and preparation of well managed core assets. By contrast, while XDDP is a less burden process which focuses only on the portion to be changed in the new system, it never prevents software structure from corrupting due to absence of the global view of the system. The paper describes an adoption process for SPL, with an example of the autonomous tractor, that applies XDDP initially for addition and modification of functions, accumulates core assets and cultivates a global view of the system through iterated development with XDDP, and finally shifts to SPL development.

Keywords
Software Evolution, Software Development Process, Software Product Line, Derivative Development

1 INTRODUCTION

The agricultural vehicle has been getting to provide more operator friendly services. Its evolution toward the unmanned vehicle is a definite trend and its final goal should be the autonomous robot with intelligence. In this forecasted evolution, embedded software controlling agricultural vehicles and robots will play an important role more than before. Most of intelligent and attractive functions to automate agricultural tasks are implemented mainly by software. These functions must analyse data from various sources including on-board sensors, GPS, other vehicles or robots, base stations, databases *etc*.; make sophisticated decisions; and drive multiple mechanical devices such as engine, brake, various implements, *etc*. in a coordinated manner through networks such as CAN, ISOBUS *etc*. Software implementing these functions grows quite easily in size and complexity. In fact, for the decade and more, automotive industry has experienced steep increase in size and complexity of software brought by integrated functions. According to Broy, 2006, in the general passenger vehicle, more than 2,000 functions were controlled by software; the size of the source code was over ten million lines; and 50 to 70% of development cost was dedicated for software. It is almost impossible to construct correctly working software of such large scale by code centric development without well-defined sound process.

Besides size and complexity, variability can be a big issue in agricultural vehicles and robots. Agricultural vehicles and robots perform different tasks for different crops under different geographical, climatic, and economic environments. They employ different technologies, namely hardware and mechanical devices, and the technologies themselves will evolve. These diversities in agriculture and technology finally result in a huge amount of variability in software.

Basically, repeated additions and modifications are applied to the existing software in evolution of software. As *ad hoc* additions and modifications are repeated, software is structurally corrupted and become low performing, resource consuming and unreliable.

This paper discusses evolutional development of software. The authors propose to introduce *software product line* (*SPL*) [Clements & Northrop, 2001; Pohl *et al.*, 2005], a paradigm of software

reuse for different products, for steady evolution without corruption of software structure. However, it is often difficult to adopt SPL without preparation even for development sites having concrete development processes for single product development. Moreover, SPL requires a global view of the current and future agricultural vehicles and robots, which is difficult to foresee for a long term. Therefore, the authors also propose to perform some iterations of the *extreme derivative development process* (or *XDDP* for short) [AFFORD] until SPL development gets applicable.

This paper is organized as follows: Section 2 gives some fundamental concepts on SPL. Section 3 describes XDDP in comparison with SPL. Section 4 shows our ideas on evolutional development starting from XDDP and shifting toward SPL. Finally, Section 5 concludes the paper.

2 PARADIGM OF SOFTWARE PRODUCT LINE

SPL development enables production of various software systems with different functionality and quality, namely software product line, in a strategic and planned manner by optimally constructing and reusing core assets shared among the systems.

SPL is absolutely not a development method to produce different products by using libraries, which store codes reusable in other products, in an *ad hoc*, code centric, individual skill dependent manner. SPL development is driven by business and technical plans of the product line. Essential plans are the scope and the road map of the product line that define which products are in and out of the product line at a certain time. Artefacts steadily reused in the products in the plans are constructed and maintained as the core assets of the product line. The core assets include not only codes but also artefacts in upper sub-processes such as requirements, specifications and designs. They are reused to construct each product by a prescribed manner, not by an individual manner.

SPL is a paradigm of software reuse among different products, rather than a certain software development methodology. A lot of methodologies and case studies based on the SPL paradigm have been presented and reported by academic researchers and industrial practitioners for the last ten to fifteen years. Some fundamental concepts in the SPL paradigm, which are described below, are introduced in these works:

Separation of domain engineering and application engineering: Domain engineering is a set of activities to construct and maintain core assets for the whole product line. Application engineering is a set of activities to develop each product by reusing core assets. These are clearly distinguished in SPL development. Moreover, management to coordinate domain engineering and application engineering is also essential. Figure 1 shows an instance of the SPL development process.

Separation of commonality and variability: Commonality and variability among products are analysed in SPL development. Commonality and variability are often described in terms of *features*, which can be defined as any prominent and distinctive concepts or characteristics that are visible to various stakeholders of the system [Kang *et al.*, 1990; Lee *et al.*, 2002]. Analysed features are categorized in terms of constraint of its selection in each product and organized as a *feature model* [Kang *et al.*, 1990]. Each product is distinguished by its equipping features.

Figure 2 shows an illustrative example feature model of an imaginary autonomous tractor product line. Each node of the feature model represents a feature. A node without any decoration represents a *mandatory* feature, which should be equipped by all the products. A node with circular decoration represents an *optional* feature, which may or may not be equipped by each product. A set of nodes bundled by an arc represents *alternative* features such that one of them is alternatively equipped by each product. Regardless of its category on selection constraint, the feature is not equipped by a product if its parent feature is not equipped in the product. The edge between nodes represents semantic relationship between corresponding features. *Consists-of* relationship means that the parent feature consists of the child feature; that is, the child feature forms a part of the parent feature. *Generalization* relationship means that the parent feature is a generalized concept of the child feature. *Implemented-by* relationship means that the parent feature is realized by the child feature. Consistent selection of features on the feature model specifies a product.

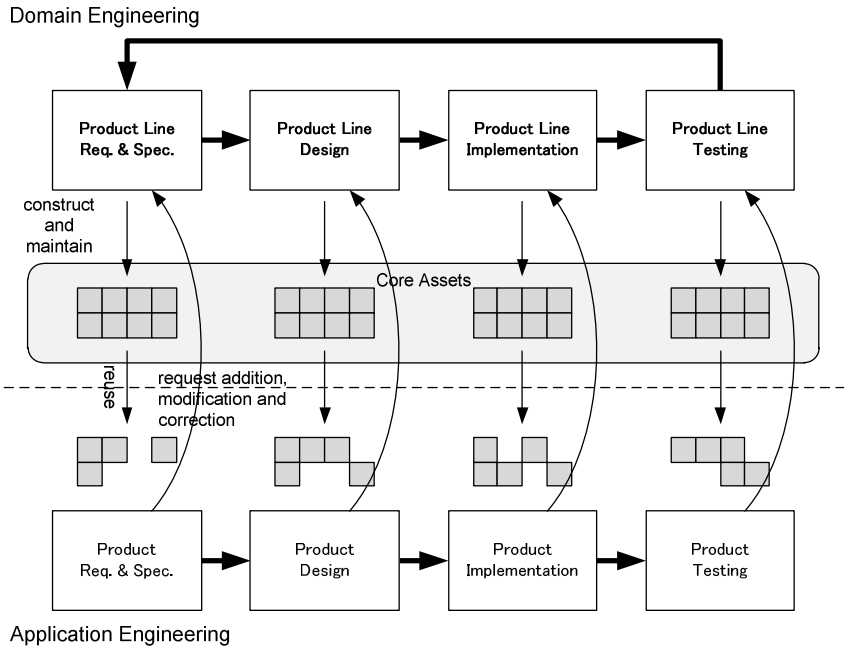

Figure 1: SPL development process

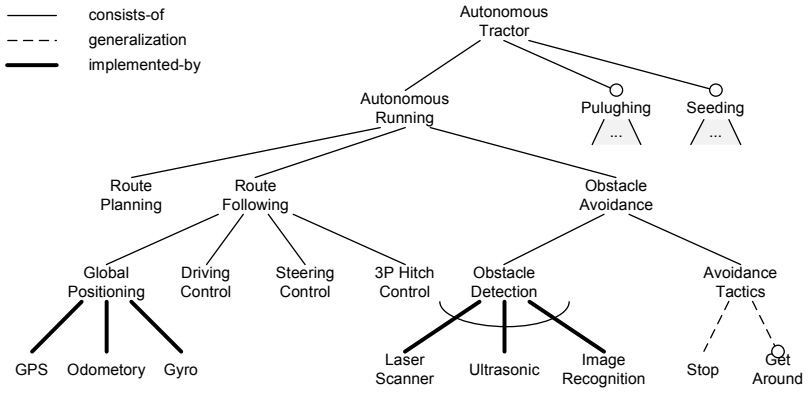

Figure 2: Feature Model

Architecture centric development: In SPL development, software architecture is established with considering the results of commonality and variability analysis to enable comprehensive and disciplined product derivation in application engineering. Core assets are also constructed to be applicable to the architecture. Application engineering is allowed to reuse core asset components at predefined points, referred to as *variation points*, in the software architecture in a prescribed manner.

Separation of problem and solution spaces: The problem space is a space storing variability models of the product line. The feature model is a representative artefact in this space. On the other hand, the solution space is a space storing other artefacts across various abstraction levels including requirements, specifications, designs, implementations, and testing. Traceability from the problem space to the solution space is somehow kept in SPL development. For example, requirement, specification, design, implementation and testing artefacts in the solution space are tagged by a feature at the portions in where the feature is realized. This traceability eases product derivation based on feature selection in application engineering.

Inherently, SPL is a development paradigm which requires a global and future view of the product line and definition of software architecture comprehending the whole product line. Most of development sites already have some working systems before introducing SPL. It is essential to

understand the whole system to define the software architecture of the product line. However, that makes introduction of SPL prohibitive for large and complicated systems due to excessively growing cost of domain engineering, time limitation, human resource limitation, lack of reliable documents *etc*. Moreover, it is often too difficult to foresee future evolution of the product line for innovative products such as agricultural robots.

Another adoption problem of SPL is the maturity level of the development site. At least, it is hopeless for development sites continuing code centric development to introduce SPL successfully. It requires a sound development process and documents in enough quality and quantity to perform domain engineering.

To alleviate these adoption barriers, the authors propose to introduce a derivative development process and then shift to SPL development.

3 XDDP: A DERIVATIVE DEVELOPMENT PROCESS

XDDP, which stands for *eXtreme Derivative Development Process* [AFFORD], is a derivative development process introduced to development sites in Japanese industries [Kobata, 2010]. XDDP is a development method to produce new products by adding and modifying an existing product. XDDP can be used as a development process to produce different products with commonality and variability likewise for SPL. However, XDDP is established independently from SPL and, in fact, it does not have fundamental concepts of SPL described in Section 2. For example, XDDP does not have the concept of the core asset. XDDP modifies the base product to construct a new product, instead of combining core assets. Figure 3 shows the overview of XDDP. Each circular node in the figure represents a sub process of XDDP. Due to page limitation, the details of each sub process are omitted.

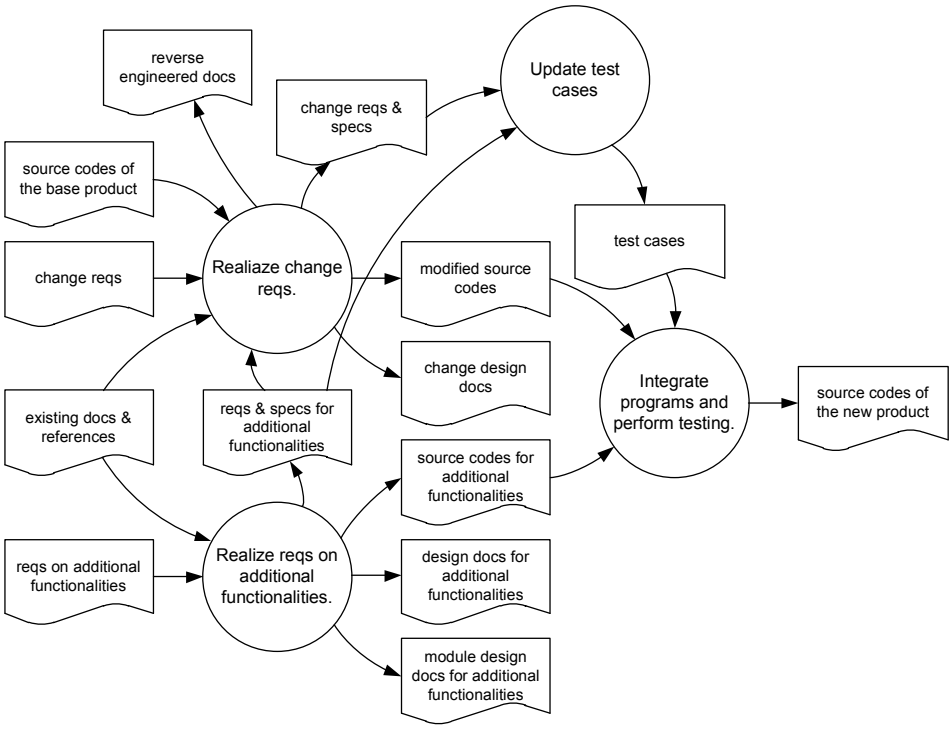

Figure 3: Overview of XDDP

XDDP starts development of a new product from describing change requirements and specifications to the base product as well as requirements and specifications for additional functions. Change requirements and specifications are different from commonly known requirements and specifications in that they regard desired changes, not desired functionality and quality, for the new product as requirements and specifications. Change requirements are description on what the stakeholder of the new product wants to be changed from the base product. Change specifications are description

specifying, namely describing without any ambiguity, how the base product should be modified to satisfy the change requirements. On the other hand, requirements and specifications for additional functions are same as commonly known requirements and specifications except they are only for additional functionalities. Note that change requirements and specifications can also include all the modifications to the base product caused by additional functions.

Table 1 shows an example of change requirements and specifications. The example is error correction in longitude and latitude observed by GPS due to tilt of the tractor. (See [Eriksen & Jæger-Hansen, 2010] for the details.) Change requirements and specifications are described with keeping their correspondence. A change requirement is followed by the change specifications satisfying it. Moreover, the change requirement is annotated by its reason to make the context of the requirement explicit.

Table 1: Change requirements and specifications

TiltComp	Change Req	Want to correct the error in longitude and latitude observed by GPS due to tilt of the tractor.
	Reason	Errors non-negligible for precise agricultural tasks are produced depending on tilt of the tractor, because the GPS antenna is attached at a distant and higher position from the reference point in the tractor.
TiltComp.1	Change Spec	Add a task to interface the inclination sensor, get the roll and pitch angles of the tractor and apply LPFs to the observed roll and pitch angles for noise reduction.
TiltComp.2	Change Spec	Let the Kalman filter in the global positioning task, which is used for better estimation of the position, use compensated longitude and latitude for its input, instead of raw longitude and latitude from GPS. Let h_{ant} be the height of the antenna, θ_{Kalman} the angle of the tractor coordinate system to the global coordinate system estimated by the Kalman filter, and θ_{roll} and θ_{pitch} the filtered roll and pitch angles respectively. The errors to real latitude and longitude due to tilt of the tractor, denoted by $\varepsilon_{\text{long}}$ and ε_{lat} respectively, are expressed as follows: $\varepsilon_{\text{long}} = h\cos(\theta_{\text{Kalman}} + \theta_{\text{P-R}})$ $\varepsilon_{\text{lat}} = h\ \sin(\theta_{\text{Kalman}} + \theta_{\text{P-R}})$ where $h = h_{\text{ant}}\sqrt{\sin^2\theta_{\text{pitch}} + \sin^2\theta_{\text{roll}}}$ and $\theta_{\text{P-R}} = \tan^{-1}\frac{\sin\theta_{\text{roll}}}{\sin\theta_{\text{pitch}}}$. These errors are added to longitude and latitude observed by GPS for compensation.

Change design documents describes necessary modification to the existing design to satisfy the change specifications, namely how the modules of the base product should be modified. The traceability matrix makes the change design documents traceable from the change requirements and specifications. Table 2 is an example of the traceability matrix after module design for the change is finished. The traceability matrix shows which module corresponding to a column should be modified to realize each change requirement or specification corresponding to a row by the check mark "X".

It will be necessary to engineer the current implementation reversely to describe change specifications and the traceability matrix, if the documents on the current implementation are not available. The results of reverse engineering are documented and referenced to describe change designs.

In the final stage, existing source codes of the base product are modified with referencing traceability matrix and change design documents to satisfy change requirements and specifications. *Ad hoc* modification of existing codes often causes newly introduced bugs. Objective of this *lazy* and

planned code modification strategy is to avoid unnecessary waste of time for repeated correction due to the bugs newly introduced by modification.

Table 2: Traceability matrix

Req/Spec ID	GetGlobalPos	GetOdometry	GetTilt [New]	GetImprovedGlobalPos	CalcSteeringAngle.	DriveSteering	…
TiltComp			X	X			
TiltComp.1			X	X			
TiltComp.2				X			

While SPL is plan driven as described in Section 2, XDDP is basically change driven. XDDP focuses only on changes to the base product. Documents are produced only for the changes. It can safely state that, although XDDP has less adoption barriers than SPL, XDDP will not prevent software structure from corrupting if it is repeatedly applied without any global and future view of the product line.

4 EVOLUTIONAL DEVELOPMENT TOWARD SPL

XDDP assumes existence of neither core assets nor software architecture. It accepts the current software architecture of the base product and modifies only the portions of the base product to be changed for the new product. Naïve derivation of new products by XDDP does not accumulate core assets, recover the software architecture, and bring reform to SPL development. For steady and affordable shift toward SPL development, the authors tailor XDDP to facilitate mining of core assets from existing artefacts and cultivate the global view through its iterations. The tailored XDDP, which is named as XDDP4SPL here, follows the process described below.

Describing requirements and specifications before and after changes: The original XDDP describes change requirements and specifications for derivation of a new product. XDDP4SPL additionally describes requirements and specifications before and after changes in separate. The *before*- requirements and specifications are imported from existing ones of the base product, or engineered reversely from the source codes of the base product if no document on requirements and specifications is available. The *after*- requirements and specifications are newly described based on change requirements and specifications.

Although readers may think that change requirements and specifications are no longer necessary, they should be kept with *before*- and *after*- requirements and specifications for some reasons.

One reason is that desire for succeeding products is often described as changes to the proceeding products at first. Moreover, the changes are described in various abstraction levels by various stakeholders of the product line. Some changes may be described by users at the abstraction level of requirement as additional or improved functions. Other changes may be described by engineers at the abstraction level of specification without explaining why the changes are needed.

Another reason is that change requirements and specifications describe why one function is newly introduced and record evolution of the product line. These documents are helpful for engineers newly involved in the project to understand the product line better than each function is explained solely.

Before- and *after*- requirements and specifications should be traceable from their corresponding change requirements and specifications. Table 3 shows a possible description. *Before*- and *after*- requirements and specifications are traceable by requirement and specification IDs in this description.

Table 3: *before*- and *after*- requirements and specifications

TiltComp	Before Req	Want to know the current position of the tractor **without** taking account of tilt of the tractor.
TiltComp	After Req	Want to know the current position of the tractor **with** taking account of tilt of the tractor.
TiltComp.1	Before Spec	
TiltComp.1	After Spec	Get the roll and pitch angles of the tractor periodically and apply LPFs to the observed roll and pitch angles for noise reduction.
TiltComp.2	Before Spec	The Kalman filter in the global positioning task uses **raw longitude and latitude** from GPS for its input.
TiltComp.2	After Spec	The Kalman filter in the global positioning task use **compensated longitude and latitude** for its input. (See Table 2 for the details of the compensation.)

Performing local variability modelling: XDDP4SPL performs local variability modelling for the limited portion of the system to be changed for a new product. With comparing *before*- and *after*- requirements and specifications, it becomes easier to identify common and different aspects such as structures, behaviours, and properties among products and define features. Different description between *before*- and *after*- requirements and specifications, which are in bold and underlined texts in Table 3, is a basis to identify features. Features indirectly related to the changes do not appear in *before*- and *after*- requirements and specifications. Instead, they may found in reverse engineered documents. Variability possibly introduced in future should be identified during local variability modelling.

The primary object of this partial feature modelling is better separation of variability, which will bring better modularization and interface design for reuse among products. Features should be identified such that commonality and variability are cleanly separated. A common feature must not include variable aspects and *vice versa*. Moreover, variable features should be orthogonally separated. A variable feature should not include multiple aspects which are in different concepts or abstraction levels. Guidelines on feature modelling [Lee, 2002] are also helpful for good feature modelling. Other objectives of local variability modelling are better understanding and intuitive representation of the portion directly and indirectly related to the changes.

Figure 4 shows an example of the local feature model. The features identified from *before*- and *after*- requirements and specifications are *Tilt Compensation* and *Inclination Sensing*. *Global Positioning* is identified as the parent feature of *Tilt Compensation* in *consists-of* relationship, since *Tilt Compensation* is for *Global Positioning*. *Kalman Filtering* is a feature identified in reverse engineered documents. It is also a sub feature of *Global Positioning* in *consists-of* relationship. Both *Kalman Filtering* and *Tilt Compensation* are modelled as optional features to enable core assets to be reused for tractors without gyro, odometory, and inclination sensors.

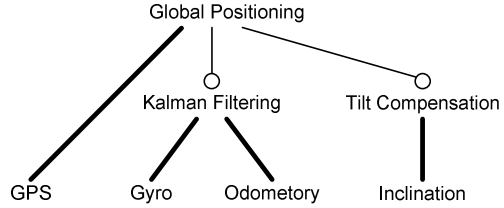

Figure 4: Local Feature Model

Describing partial requirements and specifications for the product line: Based on the *before*- and *after*- requirements and specifications, the local feature model, and reverse engineered documents, partial requirements and specifications for the product line are described. Local variability modelling and description of partial requirements and specifications can be performed iteratively. The partial requirements and specifications become core assets of the product line.

Table 4 shows an example of partial requirements and specifications. The conditional expression written in the brackets ([]) is a guard expression representing a feature selection such that the specification is activated. The term in the expression becomes true if and only if the feature of the same name is selected in the product.

Table 4: Partial requirements and specifications

GlobalPos	Req	Want to know the current position of the reference point.
GlobalPos.1	Spec	Get the current position, namely longitude and latitude, of the tractor from the GPS receiver.
GlobalPos.2	Spec	Get the roll and pitch angles of the tractor from the inclination sensor and apply LPFs to the observed angles for noise reduction. [Inclination]
GlobalPos.3	Spec	Compute errors in longitude and latitude observed by GPS to the reference point due to tilt of the tractor (See Table 2 for the details of the compensation.) and add them to the current position of the tractor from the GPS receiver for compensation. [Tilt Compensation]
GlobalPos.4	Spec	Get the direction of the tractor from the gyro sensor and apply LPFs to the observed direction for noise reduction. [Gyro]
GlobalPos.5	Spec	Get the odometory data of the tractor from the odometory sensor and apply LPFs to the observed data for noise reduction. [Odometory]
GlobalPos.6a	Spec	Input the raw or tilt compensated current position, the direction, and the odometory data of the tractor into the Kalman filter for better estimation and output the results as the current position of the reference point. [Kalman Filtering]
GlobalPos.6b	Spec	Output the results as the current position of the reference point. [!Kalman Filtering]

Performing additional design and implementation and refactoring existing artifacts: Design and implementation for partial requirements and specifications should be performed. Design and implementation for additional functionalities are constructed newly because there is no asset for them. Existing design artefacts relating to the changes are refactored if they are available, or engineered reversely from the codes otherwise. Existing codes relating to the changes are also refactored. It is essential to introduce variation mechanism, which enables product derivation by combination of core assets depending on feature selection, such as parameters, conditional compilation, common interface, inheritance, *etc*. [Anastasopoulos & Gacek, 2001; Gomaa & Webber, 2004]

Figure 5 shows the overview of XDDP4SPL. Iteration of XDDP4SPL, which is driven by changes, accumulates core assets including partial feature models, partial requirements and specifications, refactored design artefacts and codes, and reverse engineered documents. These locally mined or produced core assets should be sooner or later integrated in a global framework of the product line.

To guide this integration of core assets and facilitate shift toward SPL development, the authors present a status model of the feature. The status model defines *visible* and *invisible* features. The visible feature is one that the modeller has recognized (and thus the visible feature can be modelled in the feature model). Invisible features, which the modeller has not recognized yet, are concealed in the explored portion of the existing system. The invisible feature becomes a visible feature, when it is exposed by reverse engineering work, expert knowledge *etc*.

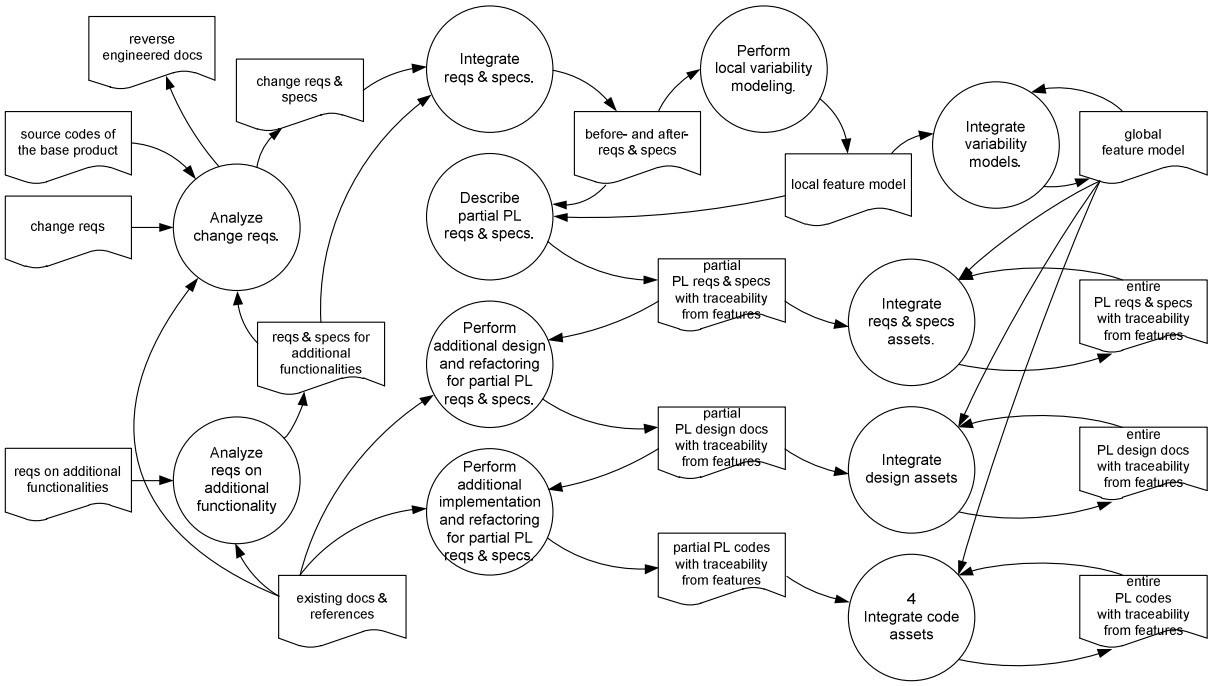

Figure 5: Overview of XDDP4SPL

Moreover, for the visible feature, the status model has two dimensions: scope of feature identification and traceability to core assets. In terms of scope of feature identification, the feature is categorized into *locally identified feature* or *globally identified feature*. The locally identified feature is one identified in a limited scope of the product line. The globally identified feature is one identified in the full scope of the product line. In terms of traceability to core assets, the feature is categorized into *core asset traceable feature* or *core asset untraceable feature*. The core asset traceable feature is one that is traceable to its related core assets. The core asset untraceable feature is one that is not traceable to its related core assets. Thus, each visible feature has four kinds of status in our model.

The feature identified in XDDP4SPL becomes a *locally identified & core asset traceable* feature, since its related artefacts are incorporated in core assets in a traceable manner. The *locally identified & core asset traceable* feature promotes to a *globally identified & core asset traceable* feature, when its position in the global feature model is determined with understanding of the whole product line.

The feature identified in the top-down manner based on expert knowledge initially becomes a *globally or locally identified & core asset untraceable* feature depending on the scope of the expert knowledge, since its related artefacts are not clear at all. The *globally or locally identified & core asset untraceable* feature promotes to a *globally or locally identified & core asset traceable* feature, when its related artefacts are incorporated in core assets in a traceable manner after XDDP4SPL iteration or reverse engineering work.

This categorization of the feature is used to separate the portion to where SPL development is applied and the portion to where derivative development is applied in the system. Shift to perfect SPL development is achieved when i) all the visible features are *globally identified & core asset traceable* and ii) invisible features are believed to be wiped out.

5 CONCLUSION

This paper presented an evolutional development process combining SPL and XDDP, which the authors referred to as XDDP4SPL. The process is as follows: i) Change requirements and specifications are described with focusing on changes to the base system as XDDP does. ii) Based on the change requirements and specifications, *before*- and *after*- requirements and specifications are described to make commonality and variability between the base system and the new system. iii) Local variability modelling, which constructs a local feature model, is performed for better separation

of variability and better understanding and intuitive representation of the portion related to the changes. iv) Partial requirements and specifications, which are incorporated in core assets, are described with establishing traceability from features. v) Existing design artefacts and codes are refactored with introducing variability mechanisms to enable product derivation by combination of core assets.

Core assets accumulated through some iterations of XDDP cultivates global view of the system and enables shift to SPL development. To guide integration of core assets and facilitate shift toward SPL development, the paper presented a status model of the feature in the existing system. The feature is categorized into invisible and visible features. The visible features is categorized into four classes, namely { *locally-identified*, *globally-identified* } × { *core asset traceable*, *core asset untraceable* }.

The future works include application and evaluation of XDDP4SPL to develop additional functions for the autonomous tractor.

ACKNOWLEDGMENT

This work was supported by Grant-in-Aid for Young Scientist (B), KAKENHI (No. 21700035).

REFERENCES

Books:

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

Pohl, K., Böckle G., and Linden, F. v. d.: *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, 2005.

Conference papers:

Anastasopoulos, M. and Gacek, C.: *Implementing Product Line Variabilities*, Proc. Symp. on Software Reusability (SSR) '01, pp.109–117, 2001.

Broy, M.: *Challenges in Automotive Software Engineering*, Proc. 28th Int. Conf. on Software Engineering, pp.33–42, 2006.

Gomaa, H. and Webber, D. L.: *Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model*, Proc. 37th Hawaii Int. Conf. on System Sciences, 2004.

Lee, K., Kang, K., and Lee, J.: *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering*, Proc. 7th Int. Conf. on Software Reuse, pp.62–77, 2002.

Kobata, K., Nakai, E., and Tsuda, T.: *Process Improvement Using XDDP: Application of XDDP to the Car Navigation System*, Proc. 5th World Congress for Software Quality, Nov. 2011.

Technical reports:

Kang, K., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S.: *Feature-Oriented Domain Analysis (FODA): Feasibility Study*, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-90-TR-222, Nov. 1990.

Thesis:

Eriksen, J. and Jæger-Hansen, C.: *GPS-Styring af Malerobot* (*GPS Control of Painting Robot*, in English), Department of Electrical Engineering, Technical University of Denmark, Aug. 2010. (in Danish)

Links:

AFFORDD: Association for Facilitation of Rational Derivational Development, http://www.xddp.jp, last accessed on October 10, 2011. (in Japanese)