Vol.2013-SLDM-160 No.9
Vol.2013-EMB-28 No.9
2013/3/13

派生開発からプロダクトライン開発への漸次的移行プロセス XDDP4SPLにおけるコア資産管理手法

中西 恒夫[1,a)]　久住 憲嗣[1,b)]　福田 晃[1,c)]

概要：XDDP4SPL は，派生開発の中で現有ソフトウェア資産に関する理解を深め，コア資産の新規開発や現有ソフトウェアからの発掘を進め，コア資産の蓄積を進め，プロダクトライン開発に漸次的に移行するべく提唱された開発プロセスである。その過程においては，ソフトウェアプロダクトラインのパラダイムに基づく開発と派生開発による開発が併走し，現有ソフトウェア資産については，プロダクトライン全体の中での位置づけが定まっている部分とそうでない部分，コア資産として管理される部分とそうでない部分とが併存するため，プロダクトライン開発への移行，コア資産の蓄積の現況の把握が欠かせない。この目的のため，現有ソフトウェア資産の理解の深化にあわせて，徐々にトレーサビリティ管理の詳細度を向上できる，階層化トレーサビリティマトリックスの記法，ならびにフィーチャモデルの変化に伴う更新の方法を述べる。

キーワード：派生開発，ソフトウェアプロダクトライン，XDDP，SPL，フィーチャモデル，階層化トレーサビリティマトリックス

Core Asset Management in XDDP4SPL Gradual Migration Process from Derivative Development to Software Product Line

Abstract: XDDP4SPL is a development proess to migrate gradually from derivative development to software product line development. The process faciitates gradual understanding of the existing software artifacts as well as gradual construction of core assets by additional development and mining from the existing software artifacts. During the migration to software product line development by XDDP4SPL, some development works are performed with software product line paradigm and but not for other works; some software artifacts are located in the recovered software architecture but not for other artifacts; and some software artifacts are managed as core assets but not for other artifacts. Therefore, it is essential to monitor the current status of migration to software product line development and addumulation of core assets. For this purpose, this article proposes the hierarchical traceability matrix, its format, and a way of its maintenance. They are designed to allow developers to refine precision of traceability records as the developers understand the existing software artifacts more.

Keywords: derivative development, software product line, XDDP, SPL, feature model, hierarchical traceability matrix

1. はじめに

同一製品系列でソフトウェア資産を共有，再利用し，高品質，低コスト，短納期での連続した製品開発を図るソフトウェアプロダクトライン（SPL: Software Product Line）[1], [2] のパラダイムが我が国の開発現場の関心を惹くようになって久しい。しかし，残念ながら，実際に SPL の導入に躊躇なく踏み込める企業は多くはないようである。SPL の導入戦略としては，最初から SPL 開発を行う Heavy Weight Approach と，個別製品開発を続けつつ，その成果物の中からコア資産を発掘，蓄積し，SPL 開発に移行していく Light Weight Approach が知られている [3], [4]。一般

[1] 九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University
[a)] tun@f.ait.kyushu-u.ac.jp
[b)] nel@f.ait.kyushu-u.ac.jp
[c)] fukuda@f.ait.kyushu-u.ac.jp

に SPL に関心を持つような企業は過去に多くの類似製品を開発してきている。これら類似製品に関する現有ソフトウェア資産を放棄して Heavy Weight Approach で一からコア資産を構築していくことは考えられない。

Light Weight Approach による SPL 導入であっても障壁は存在する。そもそも SPL は，製品間でアーキテクチャを共通化することで，ソフトウェア資産のトップダウンでの再利用を図るパラダイムである。現有ソフトウェア資産についての全体理解がなければ，アーキテクチャを定めることができず，SPL 開発への移行が進まない。しばしばこの全体理解が SPL 導入の大きな障壁となっている。ソフトウェアがすでに肥大化，複雑化していて全体理解に途方もない時間と費用を要したり，ソースコードのみが残っていて全体理解を助けるドキュメントが残されていなかったり，全体を理解していた古参の技術者がすでに転退職していたりするためである。

SPL のみならず，短納期での派生製品の開発を目的とする開発プロセス XDDP（Extreme Derivative Development Process）[5] も多くの類似製品を開発する企業の注目を集めている。SPL が現有ソフトウェア資産の全体理解を要求する全体最適指向の再利用パラダイムであるのに対して，XDDP は現有ソフトウェア資産に対する追加，変更の影響を受ける部分に限っての部分理解でよしとする局所最適指向の再利用プロセスである。当然ながら，XDDP の導入障壁は SPL よりも低く，実際，我が国の組込みソフトウェア産業界において多くの XDDP の実践例が報告されている。

導入障壁の低さは魅力的ではあるものの，XDDP は決して現有ソフトウェア資産からのアーキテクチャの回復や再利用可能なコンポーネントの発掘，蓄積を促すものではない。度重なる追加，変更に伴うアーキテクチャの劣化と保守性の低下を XDDP で防ぎきることは難しい。（もっとも XDDP の導入によって場当たり修正を禁じ，アーキテクチャの劣化を遅らせることは可能であろう。）そもそも XDDP にはコア資産の概念はなく，既存製品に対する追加と変更によって派生製品を開発する，コンポーネント単位ではなく既存製品全体での再利用が行われる。新しい製品の導出に要するコスト面では，一般的には，管理された製品間の共有ソフトウェア資産（コア資産）を組み合わせ，究極的には製品の導出自動化を図る SPL のほうが有利であろう。以上に述べた SPL と XDDP の長所，短所を鑑み，著者らは XDDP から SPL への漸次的移行を促す開発プロセス，XDDP4SPL を提唱している [6], [7]。

XDDP4SPL は，XDDP の実施で得られる変更要求仕様書，追加要求仕様書，スペックアウト資料（既存製品の追加，変更に関係する部分をリバースエンジニアリングした資料）に基づいて，既存製品の追加，変更にかかる部分に限定したフィーチャモデル [8]，ならびにコア資産の新規開発と現有ソフトウェア資産からの発掘を行う。さらに，こうして派生製品の開発のたびに得られる部分的なフィーチャモデルとコア資産の統合を進め，これらのプロダクトライン全体の中での位置づけを定め，派生製品の開発において SPL が適用される部分を拡大していくことで，派生開発から SPL へのソフトランディングを図る。その過程においては，SPL のパラダイムに基づく開発と XDDP による開発が併走し，現有ソフトウェア資産については，プロダクトライン全体の中での位置づけが定まっている部分とそうでない部分，コア資産として管理される部分とそうでない部分とが併存する。そのため，ソフトウェア資産の部分部分がどのような状態にあるのか管理しておく必要がある。また，SPL においてはフィーチャからコア資産へのトレーサビリティ，SPL 移行途上においてはフィーチャからこれからコア資産化される現有ソフトウェア資産へのトレーサビリティ確保が必要となるが，最初からその精度を高く，粒度を細かくすることは SPL 導入のハードルを著しく上げることとなる。まずは精度を求めず，粗い粒度でのトレーサビリティ確保を実現し，現有ソフトウェア資産の理解が進み，コア資産の改変が収束するにつれて，必要な程度まで精度を上げ粒度を細かくしていく戦術を採る必要がある。総括すると，XDDP から SPL への移行を成功裏に果たすためには開発管理，特に現有ソフトウェア資産（コア資産化されている部分，されていない部分の双方を含む）とトレーサビリティの管理が肝要となる。

本稿では，XDDP4SPL におけるコア資産管理の枠組みを提案する。なお，本稿の提案内容は，文献 [6], [7] で提案した XDDP4SPL を拡張するものとなっている。本稿 2 節では XDDP4SPL について概説し，3 節では XDDP4SPL におけるコア資産管理手法，特に階層化トレーサビリティマトリックスとその記法，保守の方法を提案し，最後に 4 節においてまとめを述べる。

2. SPL 漸次導入派生開発プロセス：XDDP4SPL

著者らの提案する XDDP4SPL は，既存製品に対する追加，変更のための幾度かの XDDP の実施を通して，現有ソフトウェア資産に対する部分理解，フィーチャモデルの部分的構築，コア資産の発掘と蓄積を重ね，最終的にはそれらを統合してプロダクトラインの全体理解を築き SPL への完全移行を図る開発プロセスである [6], [7]。図 1 に XDDP4SPL のプロセスフロー図（PFD: Process Flow Diagram）[5] を示す。

XDDP4SPL において，XDDP に対して追加，拡張された工程は以下の通りである。

- **要求仕様書の統合**：XDDP で作成される変更要求仕様書の記述内容を変更前要求仕様と変更後要求仕様に分離する。分離に際しては，変更要求仕様書と変更前／変更後要求仕様書の間のトレーサビリティを確保する

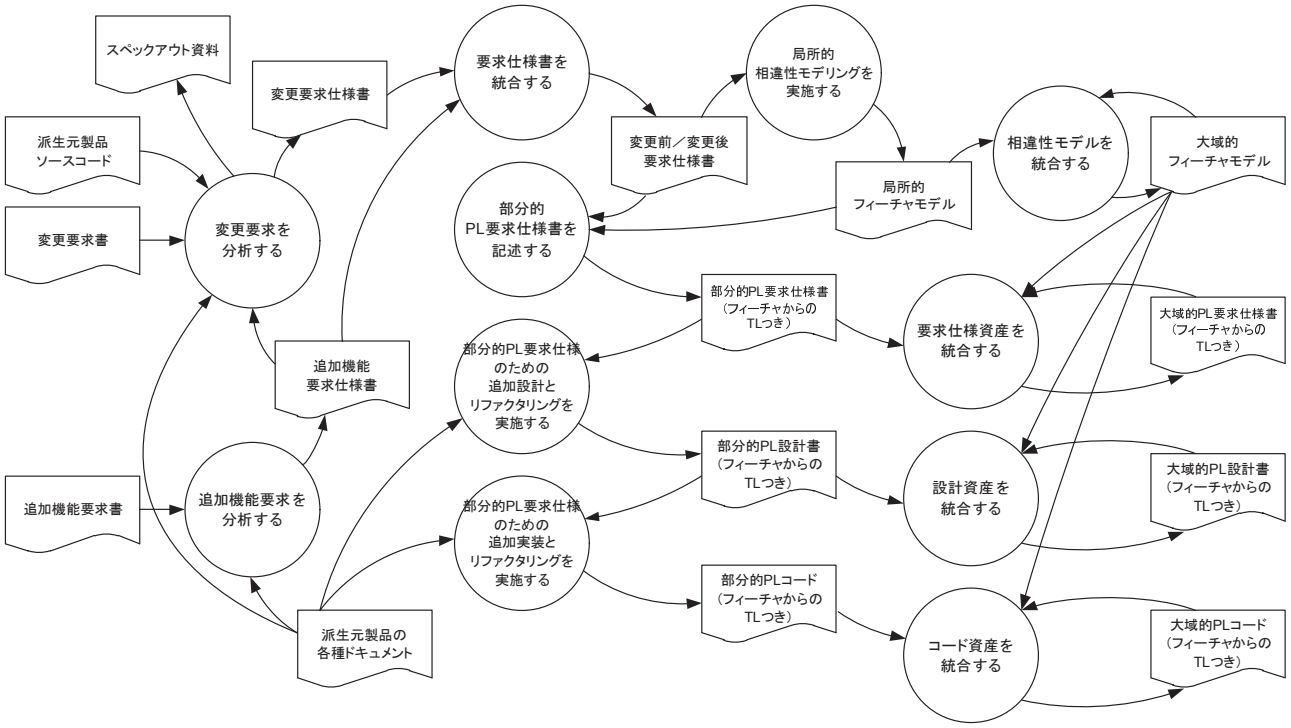

図 1 XDDP4SPL の概要（注： PL = Product Line，TL = Traceability Link）

ようにする。

- **局所的相違モデリングの実施**：既存製品に対する追加，変更にかかる部分に限定したフィーチャモデリングを実施し，局所的フィーチャモデルを構築する。フィーチャモデリングの実施にあたっては，変更前要求仕様書と変更後要求仕様書を比較し，既存製品と新製品の構造面，振舞い面，諸属性の共通点と相違点に着目する。
- **部分的プロダクトライン要求仕様書の記述**：変更前／変更後要求仕様書，局所的フィーチャモデル，スペックアウト資料を参考にして部分的プロダクトライン要求仕様書を記述する。記述に際しては局所的フィーチャモデルと部分的プロダクトライン要求仕様書の間のトレーサビリティを確保するようにする。作成された部分的プロダクトライン要求仕様書はコア資産として管理される。
- **部分的プロダクトライン要求仕様に基づく追加設計と設計変更**：部分的プロダクトライン要求仕様書に基づいた設計を実施する。追加要求については設計資産を新たに作成する。変更要求については，既存設計資産があるならばそれを再利用し，なければコードからリバースエンジニアリングして設計資産を作成する。このとき必要に応じて設計の変更を行う。設計に際しては局所的フィーチャモデルと部分的プロダクトライン設計書の間のトレーサビリティを確保するようにする。作成された部分的プロダクトライン設計書はコア資産として管理される。
- **部分的プロダクトライン設計に基づく追加実装と実装変更**：部分的プロダクトライン設計書に基づいた実装を実施する。要領は上と同じである。
- **相違性モデル，要求仕様，設計，コード資産の統合**：派生製品開発のたびごとに作成される局所的フィーチャモデル／プロダクトライン要求仕様書／プロダクトライン設計書／プロダクトラインコード資産を統合し，これらのプロダクトライン全体の中での位置づけを定め，それぞれ大域的フィーチャモデル／プロダクトライン要求仕様書／プロダクトライン設計書／プロダクトラインコード資産とする。これらの大域的資産はコア資産として管理される。統合上必要ならば，これら局所的資産，ならびに現有の大域的資産の変更を行う。

3. XDDP4SPL におけるコア資産管理

XDDP4SPL では，SPL のパラダイムに基づく開発と XDDP による開発が併走し，ソフトウェア資産についてはコア資産として管理される部分とそうでない部分，コア資産については全体の中での位置づけが確定しているものとそうでないものとが混在する。また，フィーチャからコア資産への追跡可能性の精度と粒度も時点時点で異なる。XDDP4SPL では，フィーチャ単位で SPL への移行の程度とコア資産，非コア資産を含むソフトウェア資産の現状を把握し，SPL への移行計画の策定に役立てる。

著者らは，上述の目的のためにフィーチャのカテゴリ化を行うことを，すでに文献 [6], [7] ですでに提案している。

3.1 大域識別フィーチャと局所識別フィーチャ

XDDP4SPL では，フィーチャを大域識別フィーチャと局所識別フィーチャとに大別していう。大域識別フィーチャはプロダクトライン全体の中での位置づけが明らかになっているフィーはであり，局所識別フィーチャはそうでないフィーチャである。別の言い方をすれば，大域識別フィーチャは大域的フィーチャモデル中のフィーチャであり，局所識別フィーチャは局所的フィーチャモデル中のフィーチャである。

XDDP4SPL では，派生元製品に対する追加，変更要求にかかる部分に限定して，現有ソフトウェア資産をリバースエンジニアリングし，局所的相違性モデリングを実施し，局所的フィーチャモデルを構築する。この局所的フィーチャモデル中のフィーチャは，現有ソフトウェア資産による裏付けは得られているものの，プロダクトライン全体での位置づけは明らかになっていない局所識別フィーチャとなる。派生製品開発のたびに現有ソフトウェア資産に対して理解の及ぶ範囲が広がり，局所フィーチャモデルの統合が進んでいく。局所識別フィーチャに関わる部分のアーキテクチャが明らかになれば，当該フィーチャは大域的フィーチャモデルに移管されたり，あるいは当該フィーチャを含むフィーチャモデルが大域的フィーチャモデルに統合され，当該フィーチャは大域識別フィーチャに変わる。

一方，フィーチャには現有ソフトウェア資産からボトムアップ的に見い出されるものだけでなく，プロダクトラインやそのドメインの知識に基づいてトップダウン的に見い出されるものもある。こうしたフィーチャは定義されたときから大域識別フィーチャとなる。

3.2 階層的トレーサビリティマトリックスの導入

XDDP では，トレーサビリティマトリックスを用いて変化要求あるいは変化仕様からソフトウェア資産へのトレーサビリティを確保する。トレーサビリティマトリックスは，行に変化要求あるいは変化仕様を，列にモジュールを対応させ，変化要求／仕様の実現にあたって影響を受けるモジュールにマークをつけた表である。

SPL におけるコア資産管理では，アプリケーションエンジニアリング時の製品導出を容易にすべく，フィーチャからその実現に寄与するコア資産部分へのトレーサビリティの保証が重要となる。SPL では，要求に始まり，仕様，設計，実現に至るまで様々の工程で作成される様々の抽象度の成果物がコア資産の対象となり得る。またその表現形式は，i) 自由形式の自然言語記述，ii) 箇条書きや USDM といった形式性のある自然言語記述，iii) 形式仕様記述言語による記述，iv) UML の各種図面やデータフロー図，ブロック図といったモデル図記述，v) 各種プログラミング言語によるコードなど，やはり様々のものが考えられる。しかし，抽象度，表現形式によらず，多くの場合，コア資産化の対象となる成果物は，要素に分割することができ，より抽象度の高い要素がより抽象度の低い要素を包含する階層構造を有している。たとえば，C 言語のコードには，プログラム全体 → ファイル → 関数や大域変数，型等の大域オブジェクト → 関数内の局所オブジェクトといった階層構造が見られる。

フィーチャからより抽象度の低い，細かな粒度のコア資産要素へのトレーサビリティを保証するようにすれば，当該フィーチャの実現に寄与するコア資産部分をより高い精度で切り出すことが可能となる。しかし，同時に，細粒度でのトレーサビリティ保証に要する手間は膨大なものとなる。細部をよく理解できていない箇所については当該箇所の理解に手間取ってトレーサビリティマトリックスの記述が一向に進まないし，変更が頻繁に生じている箇所については変更のたびに緻密なトレーサビリティマトリックスの修正をしなければならなくなる。この手間は派生開発から SPL への迅速な移行を妨げる要因となる。

筆者らは文献 [6], [7] においてトレーサビリティ保証の粒度については論じていなかったが，本稿では，階層的トレーサビリティマトリックスを導入し，現有ソフトウェア資産に対する理解度にあわせて，フィーチャから様々な粒度のコア資産要素へのトレーサビリティ保証を行うことを新たに提案したい。表 1 に階層化トレーサビリティマトリックスの例を示す。（例は ET ロボコン [9] の倒立型ライントレースカーの制御ソフトウェアである。）

階層化トレーサビリティマトリックスの行にはフィーチャ，列にはコア資産要素が対応する。コア資産要素の階層構造にあわせて列は階層化される。たとえば，表 1 の階層化トレーサビリティマトリックスは，フィーチャと C 言語のコード資産との間のトレーサビリティを保証するものであるが，ファイル → 大域オブジェクト（関数，大域変数，型）の階層構造にあわせて列が階層化されている。フィーチャの実現にコア資産要素が寄与するかどうか不明のとき，対応するセルには「−」が記載される。フィーチャの実現にコア資産要素が全く寄与しないことが確定されたとき，対応するセルは空欄とされる。フィーチャの実現にコア資産要素の一部が寄与することが確定されたときは「+」，コア資産要素の全体が寄与することが確定されたときは「++」が記載される。現有ソフトウェア資産に関して何も理解できていない状態では，階層トレーサビリティマトリックスのすべてのセルが「−」となっているが，現有ソフトウェア資産の理解が進むにつれ，「−」のセルが減り，空欄，「+」，「++」のセルが増えていく。また，現有ソフトウェア資産の理解が深化されるにつれ，細粒度のコア資産要素に関するセルから「−」の記載が減っていく，すなわちより精密なトレーサビリティ保証が行われるようになる。

Vol.2013-SLDM-160 No.9
Vol.2013-EMB-28 No.9
2013/3/13

表 1 階層化トレーサビリティマトリックス

ファイル / 関数 大域変数 / フィーチャ	motorctrl.c					accelsensor.c				lightsensor.c						linetrace.c				
		SetSpeed	SetDirection	GetRRotCount	GetLRotCount		SetFilteringTimeWidth	GetFilteredValue	GetRawValue		SetFilteringTimeWidth	SetBWThreshold	GetBW	GetFilteredValue	GetRawValue		SetSpeed	SetPIDParam	SetLineTraceMode	DoLineTrace
倒立走行	+	–	–	–	–	+	–	–	–											
ライントレース	+	++	++	–	–					+	+	–	–	++	++	++	++	++	++	++
マーカ検出										+	+	–	–	++	++					

3.3 階層化トレーサビリティマトリックスの保守

フィーチャからコア資産へのトレーサビリティ保証をする都合，フィーチャは一度定義されれば以後一切変更されず，既述の階層化トレーサビリティマトリックスに一切の変更が出ないことが理想型である。しかし，現有ソフトウェア資産の全体理解が十分にできていない，さらにはプロダクトラインやそのドメインに関する知識に乏しい段階では，新たなフィーチャが発見されたり，フィーチャの意味やフィーチャ間の関係が変更されたりすることが頻繁に生じる。こうしたフィーチャの変更が生じた場合に，どのように階層化トレーサビリティマトリックスの更新を行うかを以下に述べる。

フィーチャに対する原始的操作として，フィーチャの定義（定義直後のフィーチャの意味は 0 とする），削除，名称変更，意味追加，意味削除を考える。これらの原始的操作に伴う階層化トレーサビリティマトリックスの変更は以下の通りとなる。

フィーチャの定義：フィーチャを新規に定義する場合は，階層化トレーサビリティマトリックスに当該フィーチャに対応する行を設け，すべてのセルに「–」を記入する。

フィーチャの削除：フィーチャを削除する場合は，階層化トレーサビリティマトリックスの当該フィーチャに対応する行を削除するのみでよい。

フィーチャの名称変更：階層化トレーサビリティマトリックスに記載しているフィーチャの名称を変更するのみでよい。単なるフィーチャの名称を変更するだけであり，フィーチャの意味の変更はないため，トレーサビリティマトリックスのセルについては一切更新する必要はない。

フィーチャの意味追加：フィーチャ f の現在の意味を f_{SEM} とし，意味 Δs を追加するものとする。階層化トレーサビリティマトリックスの当該フィーチャに対応する行の記載を以下のように変更する。

- 「++」のセル→「++」：これまでこのセルに対応するコア資産要素全体が f の実現に寄与しており，Δs の意味の追加があってもそれは変わらない。
- 「+」のセル→「+」（または「++」）：もっとも安全な変更は「+」とすることである。このセルに対応するコア資産要素のこれまで f の実現に寄与していなかった部分がすべて Δs に関係することが確認された場合のみ「++」に変更する。
- 空欄のセル→「–」（または空欄，「+」，「++」）：もっとも安全な変更は「–」とすることである。このセルに対応するコア資産要素の一部，または全部が Δs に関係することが確認された場合，それぞれ「+」，「++」に変更する。逆にこのセルに対応するコア資産要素が Δs とまったく無関係であることが確認された場合には空欄とする。
- 「–」のセル→「–」（または空欄，「+」，「++」）：もっとも安全な変更は「–」とすることである。このセルに対応するコア資産要素の一部，または全部が f の実現に寄与することが確認された場合，それぞれ「+」，「++」に変更する。逆にこのセルに対応するコア資産要素が f の実現にまったく寄与しないことが確認された場合には空欄とする。

フィーチャの意味削減：フィーチャ f の現在の意味を f_{SEM} とし，意味 $\Delta s (\subset f_{\mathrm{SEM}})$ を削除するものとする。階層化トレーサビリティマトリックスの当該フィーチャに対応する行の記載を以下のように変更する。

- 「++」のセル→「–」（または空欄，「+」，「++」）：もっとも安全な変更は「–」とすることである。このセルに対応するコア資産要素の一部，または全部が Δs に関係することが確認された場合，それぞれ「+」，空欄に変更する。逆にこのセルに対応するコア資産要素が Δs とまったく無関係であることが確認された場合には「++」とする。
- 「+」のセル→「–」（または空欄，「+」）：もっとも

安全な変更は「+」とすることである。このセルに対応するコア資産要素のこれまで f の実現に寄与していた部分の一部，または全部が Δs に関係することが確認された場合，それぞれ「+」，空欄とする。

- 空欄のセル→空欄： これまでこのセルに対応するコア資産要素全体が f の実現に寄与しておらず，Δs の意味の削減があってもそれは変わらない。
- 「−」のセル→「−」（または空欄，「+」，「++」）： もっとも安全な変更は「−」とすることである。このセルに対応するコア資産要素の一部，または全部が f の実現に寄与することが確認された場合，それぞれ「+」，「++」に変更する。逆にこのセルに対応するコア資産要素が f の実現にまったく寄与しないことが確認された場合には空欄とする。

フィーチャ，あるいはフィーチャモデルの変更は，上述のフィーチャに対する原始的操作の組み合わせと考え，階層化トレーサビリティマトリックスの変更を行う。

4. まとめ

XDDP4SPL[6], [7] は，派生開発プロセス XDDP[5] を拡張し，派生開発の中で現有ソフトウェア資産に関する理解を部分理解から全体理解に向けて深め，コア資産の開発，発掘，蓄積を進め，SPL 開発に漸次的に移行する開発プロセスである。SPL 開発への移行の途上においてはプロダクトライン開発への移行，つまりはコア資産の蓄積の現況把握が欠かせない。この目的のために本稿では階層化トレーサビリティマトリックスを提案した。

トレーサビリティマトリックスはフィーチャの実現にどの現有ソフトウェア資産が寄与しているかを示す表である。階層化トレーサビリティマトリックスではソフトウェア資産の階層構造が表に反映されており，フィーチャから異なる抽象度のソフトウェア資産へのトレーサビリティが記述できる。そのため，現有ソフトウエア資産への理解が進んでいない段階では高い抽象度での粗いトレーサビリティを，理解が進むにつれて低い抽象度での精密なトレーサビリティの記述ができるようになっている。また，階層化トレーサビリティマトリックスでは，フィーチャの実現にソフトウェア資産の全体が寄与しているのか（++），部分が寄与しているのか（+），全く寄与していないのか（空欄），寄与しているかどうか不明なのか（−）を区別して記述できる。不明記述（−）の導入によって現有ソフトウェア資産の理解の程度が一目瞭然で把握できるようになっている。現有ソフトウェア資産の理解が進めば，それにあわせてフィーチャモデルも変更され，階層化トレーサビリティマトリックスの変更が必要となる。本稿では，階層化トレーサビリティマトリックスの保守の手順も述べた。

謝辞　本研究は科研費（No. 24500052）の助成を受けている。

参考文献

[1] P. Clements and L. Northrop, *Software Product Lines: Practice and Patterns*, Addison-Wesley, 2001.
[2] K. Pohl, G. Böckle, and F. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, 2005.
[3] P. Clements, “Being Proactive Pays Off,” *IEEE Software*, Vol.19, No.4, pp.28 and 30–31, Jul./Aug. 2002.
[4] C. Krueger, “Eliminating the Adoption Barrier,” *IEEE Software*, Vol.19, No.4, pp.29–31, Jul./Aug. 2002.
[5] 清水 吉男,『派生開発』を成功させるプロセス改善の技術と極意, 技術評論社, 2007 年 11 月.
[6] Tsuneo Nakanishi, Claes L. Jæger-Hansen, and Hans-Werner Griepentrog, “Evolutional Development of Controlling Software for Agricultural Vehicles and Robots,” *Proc. 3rd Int. Conf. on Machine Control and Guidance (MCG 2012)*, pp.101–111, Mar. 2012.
[7] 中西 恒夫, ハンス・ヴェルナー グリーペントローク, クラエス イェーガー・ハンセン, 久住 憲嗣, 福田 晃,「派生開発方法論 XDDP からのプロダクトライン開発導入」, 信学技報, Vol.112, No.23, SS2012-1, pp.1–6, 2012 年 5 月.
[8] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-Oriented Domain Analysis (FODA): Feasibility Study,” Technical Report, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-90-TR-222, Nov. 1990.
[9] ET ロボコン, `http://www.etrobo.jp/`, （最終アクセス日: 2013 年 2 月 13 日）