SOA に基づくシステムのためのアプリケーションプラットフォームのプロダクトライン化に関する研究
ー 実行可能な仕様モデルの構築 ー

M2012MM029 岡田大輝
指導教員：野呂昌満

1　はじめに

本研究室では SOA に基づくシステムのためのアプリケーションプラットフォーム (以下, App.PF) のプロダクトライン化に関する研究 [3] を行なっている．一般に，ソフトウェアの保守性向上のためにプロダクトラインの共通性，変動性を表した仕様モデルとアーキテクチャの間の追跡性確保が重要となる．江坂 [4] は，App.PF の仕様モデルのコンポーネントとアーキテクチャのコンポーネントが多対多の関係であることを示した．

App.PF のプロダクトライン化において開発の省力化のためにプロダクトアーキテクチャ（以下，PA）の自動構築が課題に挙げられる．PA 自動構築には仕様モデルのコンポーネントとアーキテクチャのコンポーネントの対応関係に基づいたモデル変換方法を定義する必要がある．

本研究の目的は App.PF のプロダクトラインにおける PA 自動構築の支援である．モデル変換により仕様モデルのコンポーネントの組み合わせに応じた PA を自動構築する実行可能な仕様モデルを定義する．PA 自動構築により，誤りの発生しやすい手作業を削減し，開発の省力化を実現する．

モデル変換による PA 自動構築実現のために仕様モデルのコンポーネントの組み合わせに応じた PA を構築する実行可能な仕様モデルを構築する．実行可能な仕様モデルは Mattsson ら [1] の記述ルールモデルとモデル変換ルールを用いた手法を参考にする．ProductLineSoftwareEngineering における仕様モデルとして一般的な Feature Oriented Reuse Method（以下，FORM）に基づき App.PF のフィーチャモデルを定義する．FORM に基づくフィーチャモデルの記述ルールモデルを定義する．App.PF のプロダクトラインの共通部分と変動部分を Bachmann らの示す変動性の分類 [2] を参考に整理し，モデル変換ルールを定義する．開発事例からアーキテクチャ上の共通部分，変動部分を整理する．変動性の分類と FORM の層の対応関係から App.PF の共通部分，変動部分と仕様モデルの関係を整理する．FORM フィーチャモデルの記述ルールモデルにモデル変換ルールを付加し実行可能な仕様モデルを定義する．実行可能な仕様モデルに対する要求充足のために実行可能な仕様モデルに GoF デザインパターンを適用する．

過去に開発された ATM 監視システムを対象に事例検証を行ない，実行可能な仕様モデルを用いてプロダクトアーキテクチャを構築する．本研究の結果として，実行可能な仕様モデルにより仕様モデルのコンポーネントの組み合わせに応じた PA の自動構築が可能となった．

2　背景技術

2.1　SOA に基づくシステムのためのアプリケーションプラットフォームのプロダクトライン

江坂ら [3] は SOA システム開発のための App.PF のプロダクトラインを定義した．プロダクトラインアーキテクチャ（以下，PLA）は App.PF に対する非機能要求を分離し，アスペクト指向アーキテクチャとして定義された．PLA はアスペクト間の関係とアスペクトの変動部分で構成される．アスペクト間の関係を図 1 に示す．

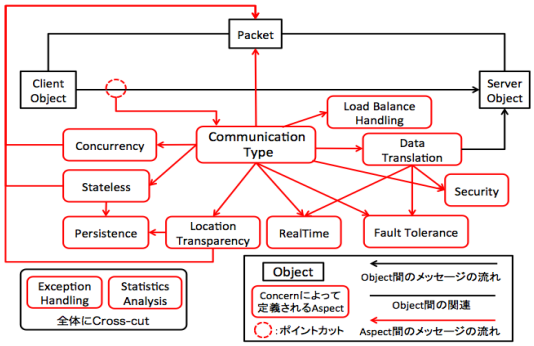

図 1　App.PF の PLA におけるアスペクト間の関係

アスペクトの変動部分は変動するコンポーネントが候補値として定義される．アスペクトのコンポーネントは仕様モデルの変動部分の候補値の選択に応じて決定する．

2.2　記述ルールモデルとモデル変換ルールを用いたモデル変換手法

Mattsson らはモデル変換によりアーキテクチャから設計を自動構築する方法を提案している [1]．アーキテクチャを設計するためのルールを機械が解釈可能な形式で記述するために UML を用いてモデル化している．

自動構築のためにアーキテクチャから設計へのモデル変換ルールを定義している．アーキテクチャ設計ルールモデルに付加した変換ルールを機械に解釈させることで設計の自動構築を実現している．設計の自動構築により誤りの発生しやすく時間のかかる手作業が削減できる．

3　App.PF のプロダクトラインの共通部分，変動部分の整理

3.1　App.PF のプロダクトラインの共通部分，変動部分とアーキテクチャの関係

App.PF のプロダクトラインの共通部分，変動部分は Bachmann ら [2] の示すソフトウェアの変動性の分類に基

いて整理した．App.PF のプロダクトラインの変動部分を表 1 に示す．

表 1　App.PF のプロダクトラインの共通部分，変動部分（一部）

変動性の分類	共通部分，変動部分	候補値	依存関係
Function	通信方式	1対1，1対多	-
Data	Message Format	SOAP,REST, Binary	Technology.Middleware. Marshalling Library
Control Flow	Instance Serialization	有,無	Function.メッセージ変換
Technology	Middleware	JBossESB, MuleESB	-
Quality Goal	Suitability	順序尺度(重要，必要，不要)	Function.送信方式
Environment	Programming Language	Java, AspectJ	-

開発事例から共通部分，変動部分とアーキテクチャの関係を整理した．例として Message Service Provider に関する変動部分とコンポーネントの関係を表 2 に示す．

表 2　Message Service Provider に関する変動部分とコンポーネントの関係

変動部分	候補値 \ Component	Message Service Provider	<<JBossESB>> Message Service Provider	<<MuleESB>> Message Service Provider	<<Service Mix>> Message Service Provider
送信方法	One To One	○	○	○	○
	One To Many	○	○	○	○
	One To One, One To Many	○	○	○	○
ESB	ESB 無	○	-	-	-
	Jboss ESB	-	○	-	-
	Mule ESB	-	-	○	-
	Service Mix	-	-	-	○

○:必要
－:関係無

3.2　App.PF のプロダクトラインの仕様モデル定義

App.PF のプロダクトラインの共通部分，変動部分を仕様上に表すためにフィーチャモデルを定義した．フィーチャモデルとしてフィーチャ図と補足情報を構築した．

変動性の分類と FORM の層の関係からフィーチャ図を構築した．共通部分，変動部分をフィーチャ，候補値をサブフィーチャとして記述した．App.PF のフィーチャ図を図 2 に示す．

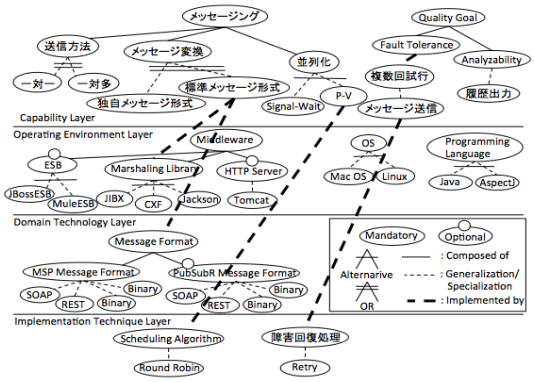

図 2　App.PF のフィーチャ図（一部）

フィーチャの補足情報を共通部分，変動部分間の依存関係から定義した．例として Marshalling Library に関する補足情報を表 3 に示す．

表 3　Marshalling Library に関する補足情報

フィーチャ	依存ルール	優先順位
JIBX	Message Format = SOAP	-
CXF	Message Format = SOAP / REST	-
Jackson	Message Format = JSON	-
Message Pack	Message Format = Binary	-
Java RMI	Message Format = RMI	-

3.3　仕様モデルとアーキテクチャの対応関係

仕様モデルのコンポーネントとアーキテクチャのコンポーネントが多対多の関係であることを確認した．フィーチャモデルのサブフィーチャの組み合わせに応じてプロダクトアーキテクチャのコンポーネントが特定できる．例としてメッセージ変換に関するコンポーネントとサブフィーチャの関係を表 4 に示す．

表 4　メッセージ変換に関するコンポーネントとサブフィーチャの関係 (一部)

Feature	SubFeature \ Component	Original Connector	<< JIBX >> SOAP Connector	<< CXF >> SOAP Connector	<< CXF >> REST Connector
メッセージ変換	メッセージ変換無	-	-	-	-
	独自メッセージ形式変換	○	-	-	-
	標準メッセージ形式変換	-	○	○	○
Message Format	Message Format 無	○	-	-	-
	SOAP	-	○	○	-
	REST	-	-	-	○
Marshalling Library	Marshaling Library 無	○	-	-	-
	JIBX	-	○	-	-
	CXF	-	-	○	○

○:必要
－:関係無

4　実行可能な仕様モデルの構築

4.1　実行可能な仕様モデルの仕様

実行可能な仕様モデルのステークホルダは要求アナリストと開発者である．要求アナリストは，App.PF に対する顧客の要求に応じて，実行可能な仕様モデル上のサブフィーチャの選択を行なう．本研究では，仕様モデルからプロダクトアーキテクチャの自動構築支援を目的としているので，ユーザの要求からサブフィーチャの選択までの作業は支援対象としない．

実行可能な仕様モデルは，要求アナリストの選択したサブフィーチャの組み合わせに応じたプロダクトアーキテクチャを構築する．サブフィーチャの組み合わせに違反があった場合は，操作 UI に違反を警告するメッセージを表示する．開発者は実行可能な仕様モデルにより構築されたプロダクトアーキテクチャを利用してプロダクトの設計，実装を行なう．

4.2　記述ルールモデル

変換対象のモデルの記述ルールを表すために FORM に基づくフィーチャモデルの記述ルールモデルを構築する．記述ルールモデルはフィーチャの種類，関係，配置される層，補足情報とそれらの関連を記述する．FORM に基づくフィーチャモデルの記述ルールモデルを図 3 に示す．

フィーチャの補足情報は Additional Information クラスとして表した．Additional Information クラスには，変動部分の候補値の決定に必要な情報のみを表した．

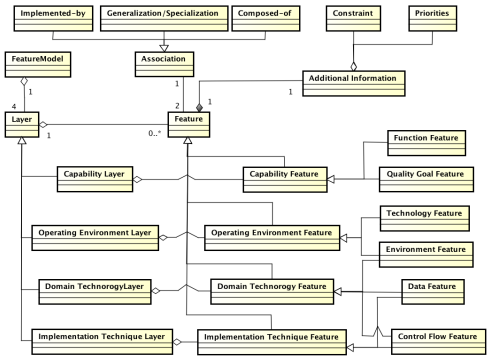

図 3　FORM フィーチャモデルの記述ルールモデル

4.3　モデル変換ルール

サブフィーチャの組み合わせに応じてプロダクトアーキテクチャが特定できるので，モデル変換ルールを仕様モデルとアーキテクチャの対応関係から定義する．サブフィーチャの組み合わせをパラメータとし，仕様モデルのコンポーネントとアーキテクチャのコンポーネントの対応関係を参照することで PA へ変換が可能となる．

4.4　実行可能な仕様モデル

複数のサブフィーチャの組み合わせに応じて PA のコンポーネントを特定する実行可能な仕様モデルを定義した．定義した実行可能な仕様モデルを図 4 に示す．

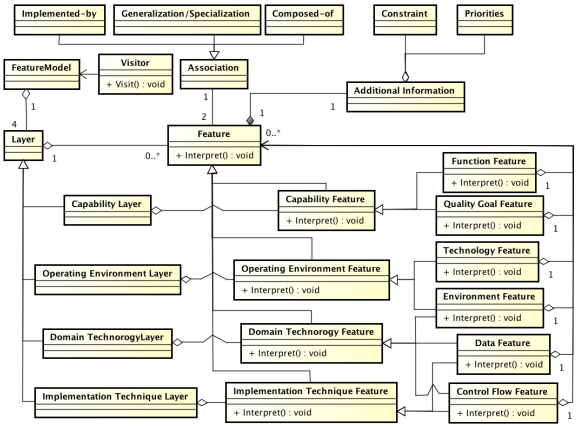

図 4　実行可能な仕様モデル

実行可能な仕様モデルに対する要求充足のために GoF デザインパターンを適用した．実行可能な仕様モデルはプロダクトの追加に応じたフィーチャの追加，変更に対する柔軟性が求められる．モデル変換を行なうために複数のサブフィーチャの組み合わせに応じた PA 特定処理の実現が求められる．

フィーチャの追加，変更に対する柔軟性向上のために，既存の構造に要素を追加するだけで追加，変更が可能となる Composite パターンを適用した．複数のサブフィーチャの組み合わせを取得するために，データ構造と処理を分離し，走査処理を実現の可能な Visitor パターンを適用した．サブフィーチャの組み合わせに応じた PA 特定処理実現のために，処理対象の意味を解析し，解析結果に応じた処理を実現する Interpreter パターンを適用した．

Visitor クラスがフィーチャモデルを走査し，各 Feature クラスの Interpret メソッドがサブフィーチャの組み合わせに応じた処理を行なう．仕様モデルのコンポーネントとアーキテクチャのコンポーネントの関係を参照することで変換後の PA を特定できる．

5　事例検証

過去に開発された ATM 監視システムを対象に，実行可能な仕様モデルを用いて PA が構築可能であることを検証する．ATM 監視システムは ATM 端末をネットワークを介して監視し，状況に応じて管理側の端末へ通知及び障害対応を行うシステムである．

ATM 監視システムの変動部分の選択を実行可能な仕様モデルへの入力とし，PA 構築を行った．メッセージ変換に関連する実行可能な仕様モデルを図 5 に示す．

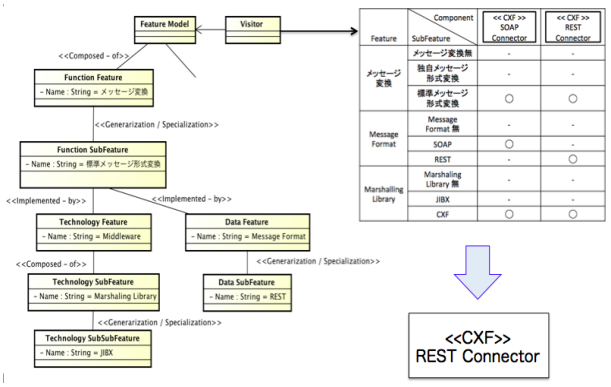

図 5　メッセージ変換に関連する実行可能な仕様モデル

Visitor がフィーチャモデルを走査し，仕様モデルとアーキテクチャの変動部分の関係を参照することでサブフィーチの組み合わせに応じた PA が構築できた．ATM 監視システムの PA を図 6 に示す．

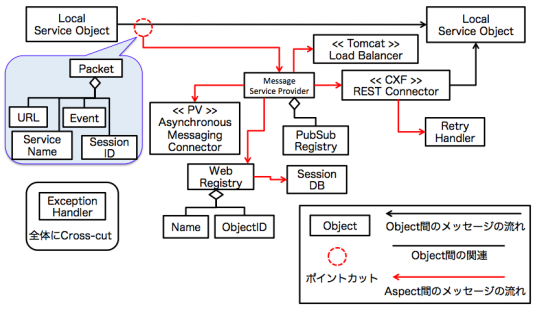

図 6　ATM 監視システムのプロダクトアーキテクチャ

6　考察

6.1　実行可能な仕様モデルを用いた PA 自動構築

伴らの研究では，App.PF のプロダクトラインの仕様モデルのコンポーネントとアーキテクチャのコンポーネントが多対多の関係であることを示した．この関係に基

づいて定義した実行可能な仕様モデルにより，複数のサブフィーチャの組み合わせに応じた PA が特定可能であることを事例検証を通して確認した．

今後の課題にプロダクトの自動構築が挙げられる．アーキテクチャから設計，実装の自動化を行なうフレームワークを構築し，実行可能な仕様モデルと統合することでプロダクト自動構築の実現が可能となる．

6.2 実行可能な仕様モデル

本研究では，実行可能な仕様モデルに対する要求充足を目的として Composite，Interpreter，Visitor パターンを適用した．各パターンに関するパターンとの比較を行ない，適用したパターンの妥当性を示す．

サブフィーチャの組み合わせに応じた PA 特定処理実現のために Visitor，Interpreter パターンを用いた．関連するパターンとして Command, Chain of Responsibility, Iterator パターンが挙げられる．複数要素の組み合わせに応じた処理を実現するパターンの比較結果を表 5 に示す．

表 5 複数要素の組み合わせに応じた処理を実現するパターンの比較

比較内容 / パターン	フィーチャの追加，変更に対する柔軟性	サブフィーチャ選択に応じた処理の実現	複数の要素の組み合わせに応じた処理の実現
Command	△	○	△
Chain of Responsibility	△	○	△
Interpreter	○	○	△
Visitor	○	-	○
Iterator	△	△	○

○:有効 △:必ずしも有効でない -:関連なし

Command パターンではフィーチャ毎に Command オブジェクトを定義し，それらを組み合わせて処理を行なうので管理が困難になる．Chain of Responsibility，Iterator パターンでは，走査順序を定義する必要がある．また，Iterator パターンではフィーチャ毎に Iterator を定義する必要があり冗長となる可能性がある．Visitor パターンでは順序を意識すること無くフィーチャモデルを走査し，複数のサブフィーチャの組み合わせる処理を実現できる．Interpreter パターンを用いてサブフィーチャの組み合わせの意味を解釈し，結果に応じて処理を行なうことで複雑さを低減できる．サブフィーチャの組み合わせに応じた PA 特定処理の実現には Visitor パターンと Interpreter パターンの適用が妥当である．

6.3 モデル変換手法

本研究では，Mattsson らの提案する記述ルールモデルとモデル変換ルールを用いたモデル変換手法により実行可能な仕様モデルを定義し，仕様モデルから PA へのモデル変換を行なった．他のモデル変換手法にモデル変換言語である Atlas Transformation Language(以下，ATL) を用いたモデル変換が挙げられる．ATL を用いたモデル変換の概要を図 7 に示す．

ATL では，対象となる入力インスタンス Ma を出力インスタンス Mb へモデル変換する．モデル変換のためには，入出力インスタンスのメタモデル MMa，MMb，メタモデル間の変換ルールを定義した MMa2MMb.atl を定義する必要がある．

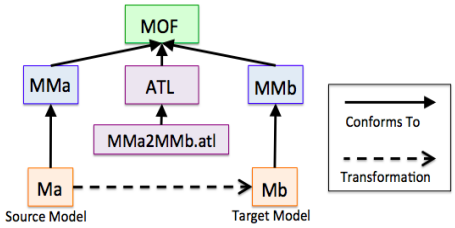

図 7 ATL を用いたモデル変換の概要

実行可能な仕様モデルは ATL よりも変換ルールの変更に対する柔軟性が高い．ATL を用いた場合，変換ルールの変更が複数箇所に及び，複雑化する可能性がある．実行可能な仕様モデルでは，サブフィーチャ，組み合わせに応じたコンポーネントの指定だけで変更が可能である．

実行可能な仕様モデルを用いて変換の記述ルールを表現することは，開発の省力化と変更に対する柔軟性．実行可能な仕様モデルは，変換対象である入力インスタンスに対して，記述ルールに従っていることをチェックし，仕様の決定からアーキテクチャ設計までに起こりうる手戻りを自動的に検出可能である．ATL を用いたモデル変換で同様の処理を行おうとした場合，前述した理由から処理が複雑化し，管理が困難となる．

7 おわりに

本研究では App.PF の仕様モデルのコンポーネントとアーキテクチャのコンポーネントの関係に基づき，PA 自動構築を行なう実行可能な仕様モデルを定義した．実行可能な仕様モデルを用いて複数のサブフィーチャの組み合わせに応じた PA が構築可能であることを事例検証により確認できた．実行可能な仕様モデルを用いることで PA 構築に要する工数の削減が可能となった．今後の課題にアーキテクチャから設計，実装の自動化を行なうフレームワークを構築することによるプロダクト自動構築が挙げられる．

参考文献

[1] A. Mattsson, B. Fitzgerald, B. Lundell and B. Lings, “ An Approach for Modeling Architectural Design Rules in UML and its Application to Embedded Software,” *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 2, 2012.

[2] F. Bachmann, L. Bass, ”Managing Variability in Software architectures, ” *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 3, pp. 126-132, 2001.

[3] 江坂篤侍, 野呂昌満, 沢田篤史, “ SOA に基づくシステムのためのアプリケーションプラットフォームのプロダクトライン化に関する研究, ” 情報処理学会研究報告. ソフトウェア工学研究報告, Vol. 2013-SE-179, no. 25, pp. 1-6, 2013.

[4] 江坂篤侍, 研究ノート.