博　士　論　文

自己適応を目的としたソフトウェアアーキテクチャの構築と運用に関する研究

D2013MM001　江坂 篤侍

指導教員　野呂 昌満

2018 年 2 月

南山大学 数理情報研究科 数理情報専攻 博士後期課程

A Study on Construct and Management for Self-Adaptive Software Architecture

2013MM001　ESAKA Atsushi
Supervisor　NORO Masami

February 2018

Graduate Program in Mathematical Sciences and Information Engineering
Graduate School of Mathematical Sciences and Information Engineering Nanzan University
Nanzan University

要約

CPU の性能ならびにネットワークの通信速度の飛躍的な向上にともない，エンタープライズシステムやインタラクティブシステムなどの使用者が対話的に操作するシステムから，組込みシステムなどの厳しい実時間制約を持つシステム，その発展として IoT システム，近年，研究が盛んに行われている人工知能技術を適用したシステムまでの多様な応用領域のソフトウェアに自己適応性が適用されている．Salehie らは自己適応計算方法を 14 の局面で分類している．既存の自己適応のための技術は特定の応用領域や計算方法に依存したものである．

本研究の目的は，自己適応ソフトウェアの作成を支援することである．多様な自己適応計算方法をアーキテクチャレベルからコードレベルまで統一的に扱う．ここで，統一的とは，自己適応計算の単一の記述法を与え，その実現 (設計) 方法を定義し，自己適応計算方法それぞれに対して実現方法の 1 つまたは複数を対応付けた枠組みによって扱うことを意味する．

構造だけでなくソフトウェアプロセス定義や PLSE(Product Line Software Engineering) 応用についても視野にいれ，アーキテクチャ中心の自己適応開発支援について考察する．自己適応のためのパターンを定義し，これを用いることによって，Salehie らの分類する自己適応計算方法を統一的に記述可能にする．異なる自己適応計算方法を統一的に扱うために，共通なメタパターンを定義する．このメタパターンから特定の計算方法に依存した (ベース) パターンを導出する．参照アーキテクチャの設計には，導出されたアーキテクチャパターンを適用し，具象アーキテクチャの設計にはデザインパターン，実装ではコーディングパターンを適用する．これにより，アーキテクチャレベルからコードレベルまで矛盾なく自己適応計算を記述可能とする．

組込みシステムにおいては，コンテキスト指向とアスペクト指向の統一的な取り扱いをより上位の概念である自己適応の問題として解決することを技術課題とする．すなわち，自己適応計算として，組込みシステムにおける全てのコンテキストとアスペクトを記述する．また，実時間制約を考慮して自己適応の実現方法を選択し，実現するための枠組みを定義する．

インタラクティブシステムにおいては，ソフトコンピューティングとハードコンピューティングの統一的な取り扱いを自己適応の適応ポリシーの問題として解決する．すなわち，コンテキストに応じて活性化させる振舞いの決定方法の問題である．使用者インターフェースの動的変更を行なうアプリケーションを事例とし，考察する．

モデルコンパイラにおいては，対象モデルの変更を考慮したモデルコンパイラの実現を，モデルおよび変換規則を入力としたメタモデルコンパイラによる適応処理の問題として解決する．すなわち，言語モデルをパラメータとしたメタモデルコンパイラのモデルフリー適応によって，モデルコンパイラの対象モデルの変更を実現する．

IoT システムにおいては，他の自己適応ソフトウェアとの相互運用を考慮した自己適応のための構造を定義することを技術課題とする．コンテキスト協調を他のソフトウェアの影響を起因とした外部適応として捉え，この外部適応の記述をメタレベルの自己反映適応記述の問題として解決する．

本研究の成果として，単一の自己適応のためのメタパターンを用いて，組込みシステム，インタラクティブシステム，モデルコンパイラ，IoT システムに求められる自己適応計算を記述できた．応用領域や計算方法によらず一般的に自己適応計算を取り扱うための枠組みが得られた．

Abstract

Domain specific software architecture defines the optimum structure of software in the domain. The application framework and The program code generator are based on the architecture.

When designing the architecture, it is important to identify the concern which is in the domain be able to decomposition as the modules without any contradiction. The architecture for interactive systems needs to be designed as explainable about all existing architectures. The architecture for embedded systems needs to be designed to consider both of the context concern and the non-functional characteristic concerns. The architecture for IoT systems needs to be designed as dynamically reconfigurable while considering non-functional characteristics. In the architecture centric development, an application framework and a program code generator are used. However, the data which is the input and output of generator are differed depends on the environment, therefore which works the various environment, it is necessary to define meta-generator.

In this paper, we define the domain specific architecture of interactive system, embedded system, and IoT system with the consideration of concerns of each systems and discuss the usefulness of these architectures. We also design the program code generator, to support the architecture centric development, and discuss the usefulness of it. We construct the common architecture for interactive systems which is a set of the common reference architecture and the common application architecture. We construct the architecture for embedded systems with consideration for treating both of considering context concern and non-functional characteristic concerns as some modules. We construct the architecture for IoT systems which can reconfigure the mobiles and the fogs. We construct the architecture for the generator which generate program code based on those architecture.

We design these architecture by introducing the aspect oriented technology. This technology make it possible to separate the concerns as aspect in the domain. We define the implementation of programming language independence is able to incarnate by using design pattern. For interactive systems, we identify the cross-cutting concerns which makes the MVC architecture separate. For embedded systems, context and nonfunctional characteristics define as aspects. For IoT systems, reconfiguration realize by cooperation between contexts. We think the generator and meta generator can be regarded as data structure conversion system, so we design common architecture of the data structure conversion system.

Finally, we evaluate the usefulness of the designed aspect-oriented domain specific architecture and summarize the results of this paper.

目次

第 1 章

序論

本章では，近年，設計・実現されている自己適応ソフトウェアにおける背景を述べる．その後，本研究の目的について述べる．

1.1 研究の背景

CPU の処理速度ならびにネットワークの通信速度の飛躍的な向上に伴い，自己適応ソフトウェアを実現するための研究が盛んに行われるようになってきた [55].

エンタープライズシステムやインタラクティブシステムなどの使用者が対話的に操作するシステムから，組込みシステムなどの厳しい実時間制約を持つシステム，その発展としての IoT システム，近年，研究が盛んに行われている人工知能技術を適用したシステムまで，多様な応用領域において自己適応ソフトウェアが実現されている．

エンタープライズシステムにおいて，自己適応ソフトウェアは，顧客情報等に応じて自己適応的に業務ロジックが変化するものとして実現され，様々な場面において利用されている．例えば，電子取引システムに対して，会員クラスに応じた購入プロセスの動的再構成 [48] などが挙げられる．

インタラクティブシステムにおいては，アプリケーションロジックだけではなく，使用性の向上を目的として使用者インタフェースの自己適応も実現されている．例えば，インタラクティブシステムでは実行時環境として多様なデバイスが提供されていることから，レスポンシブ Web デザイン [43] 技術を適用することにより，デバイスの画面サイズに応じた表示画面の動的再構成が実現される．

厳しい実時間制約を持つ組込みシステムは，スマートホーム [11]，自動販売機，自動改札機など，その振舞いが外部環境に応じて自己適応的に変化するものとして実現されている．これらは，近年のスマートデバイスの普及に伴い，移動体としてのスマートデバイス上のソフトウェアと連携する．このスマートデバイス上のソフトウェアも，その振舞いも外部環境に応じて自己適応的に変化するものとして実現する試みが盛んに行われている [18]．自己適応による耐故障処理の実現に関する試み [61] 等，様々な非機能特性が自己適応によって実現されている．

組込みシステムの発展としての IoT(Internet of Things) システムでは，移動体としての組込みシステムとサービスが連携し，これらの連携を外部環境や稼働状況に応じて自己適応的に変化するものとして実現される．例えば，実行時のサービスの通信品質 (QoS：Quality of Service) を保証するために，動的なルーティング [66] や，協調するサービスの動的な選択 [22] が実現されている．

人工知能の分野において，古くはエキスパートシステムでは，知識と事実の矛盾を解消するポリシーを，知識と事実の構成に応じて変更する試みがなされている．近年，盛んに研究されている深層学習では，学習の過程におけるシナプスの組み替えの実現に対して，自己適応アプローチを取ることも可能である．

自己適応ソフトウェアとして実現することが困難であったミドルウェアや基本ソフトウェア等の応用領域においても，今後，実現可能になると考える．例えば，JIT(Just In Time) コンパイラによる，実行時の状況に応じた最適化コンパイルが実現されている [62]．その他にも，SOA に基づくシステムでは，仮想マシンの稼働

状況に応じたライブマイグレーションを実現する試みがなされている [13]．オペレーティングシステムにおいては，外部状況や実行されるプロセスの性質に応じてスケジューリングポリシーを変更する試みがなされている [64]．

Salehie ら [55] は，自己適応計算方法を 14 の局面で分類し，この局面を次の 4 つのグループにまとめている．

1. 適用対象
2. 実現方法
3. 時制特性
4. 適応結果の影響

これらは，

a) どの対象を，
b) どのような方法で，
c) どのような実時間制約の下，
d) 自己適応を行なった結果がどのように反映されるか

によって自己適応ソフトウェアを分類する．以下，それぞれの詳細を述べる．

適用対象は，自己適応の適用対象群を中心に分類する局面のグループである．このグループは，レイヤ，粒度，コストの局面で構成される．レイヤは水平応用領域を示す．アプリケーションソフトウェアやミドルウェア，基本ソフトウェアなどを分類するための局面である．粒度は，コンポーネント，コンポーネント群，サブシステム等などの適応対象の粒度を分類する局面である．コストは，適応の実行時間，要求される資源，適応処理の複雑さ等で分類する．

実現方法は，コード記述上の取り扱い，実行時の取り扱いを中心に分類する局面である．すなわち，自己適応の起因となる事象の認識や自己適応動作を，開発時と実行時にどのように扱うかによって分類する．このグループは，動的/静的，外部適応/内部適応，ソフトコンピューティング/ハードコンピューティング，モデルフリー/モデルベース，応用領域依存/一般化の局面で構成される．動的は動的織込み，静的は静的織込みを示す．外部適応は耐故障性のように内部要因による適応，内部適応はコンテキストアウェアネスのように外部要因による適応を示す．知的な適応は，ソフトコンピューティングによって実現される．数理モデル等のモデルに基づく適応はモデルベースに分類される．応用領域依存は垂直応用領域への依存を示す．

時制特性は，時制に関する対象群の取り扱い方法によって分類するための局面である．すなわち，自己適応の起因となる対象群の監視方法と自己適応の実行を行なう時機によって分類する．このグループは，継続的/適応的監視，リアクティブ/プロアクティブの局面で構成される．継続的/適応的監視は，すなわち，ポーリングによるイベント発生検知，インタラプトによるイベント発生検知による分類のための局面である．リアクティブ/プロアクティブは，イベントが発生した後に適応を行なうものと，発生を予想して適応を行なうものに分類される．耐故障処理のための自己適応では，障害時の影響を小さくするためにはプロアクティブ適応を実現すべきであり，実時間効率および資源効率を考慮すると，リアクティブな監視方法は好ましくないと述べている．

適応結果の影響は，自己適応の結果が何に影響を与えるかによって分類する局面である．このグループは，人間機械系，適応結果の信頼性，他の自己適応ソフトウェアと相互運用の有無の局面で構成される．適応結果の信頼性は，機械的に結果の良し悪しが判断できるか，人間等による解釈が必要かどうかで分類する．他の自己適応ソフトウェアとの相互運用を考慮した自己適応ソフトウェアに関する研究成果は少ない．

特定の応用領域や計算方法に依存して，アプリケーションフレームワーク [20, 29, 46]，ミドルウェア [10]，言語 [41] 等の技術が提案されている．

1.2 目的

本研究の目的は，自己適応ソフトウェアの作成を支援することである．多様な自己適応計算方法をアーキテクチャレベルからコードレベルまで統一的に扱う．ここで，統一的とは，自己適応計算の単一の記述法を与え，その実現 (設計) 方法を定義し，Salehie らに分類される自己適応計算方法それぞれに対して実現方法の 1 つまたは複数を対応付けた枠組みを定義し，これにより自己適応計算を記述することである．組込みシステム，インタラクティブシステム，モデルコンパイラ，IoT システムの 4 つの事例研究を通して，その結果を一般化し，考察する．

組込みシステムの事例研究では，コンテキスト指向組込みソフトウェアの作成支援を目的として，アスペクト指向とコンテキスト指向を統一的に扱う方法を提案する．過去の研究成果として，組込みシステムにおいて考慮すべき非機能特性を特定し，アスペクトとして分離したアスペクト指向アーキテクチャを提案してきた [67]．前述のように近年では組込みシステムを自己適応ソフトウェアとして設計・実現する試みが盛んに行われている [18]．特に移動体としての組込みシステムは，外部環境を反映するシステムの内部状態をコンテキストとした，コンテキスト指向に基づいて実現されている．これまでの研究成果を拡張し，アスペクト指向とコンテキスト指向を統一的に扱う．

インタラクティブシステムの事例研究では，コンテキスト指向インタラクティブソフトウェアの作成支援を目的として，コンテキストによる使用者インターフェースの動的変更のための構造を定義する．この変更は，操作履歴等から使用者の嗜好を学習することによって行なう．

モデルコンパイラの事例研究では，モデルコンパイラの作成支援を目的として，モデルコンパイラおよびメタモデルコンパイラの構成法を提案する．一般に特定のモデルに特化したモデルコンパイラが提案されている．この対象モデルの変更に対応するためにメタモデルコンパイラを実現する．メタモデルコンパイラはモデルフリー適応を実現し，言語モデルと変換規則を入力としてモデルコンパイラの適応処理を行なう．

IoT システムの事例研究では，他の自己適応ソフトウェアとの相互運用を考慮した IoT システムの作成支援を目的として，IoT システムのアーキテクチャを定義する．自己適応ソフトウェアは，その適応結果によってコンテキストが変化し，この変化に応じて他の自己適応ソフトウェアの振舞いが変化する (以下，コンテキスト協調)．前述のように，コンテキスト協調を考慮した自己適応ソフトウェアに関する研究成果は少ないことから，この相互運用を考慮した自己適応ソフトウェアの構成法を定義する．

1.3 本論文の構成

第 1 章の本章では，本研究の背景と目的および技術課題を述べる．第 2 章では，本研究に関連する研究および，本研究の位置付けを述べる．第 3 章では，本研究の技術課題および研究の進め方を述べる．第 4 章では，自己適応のためのパターンとして，PBR (Policy-Based Reconfiguration) パターンを提案する．第 5 章では，組込みシステムのアーキテクチャを提案し，この有用性について述べる．第 6 章では，インタラクティブシステムのアーキテクチャを提案し，この有用性について述べる．第 7 章では，モデルコンパイラおよびメタモデルコンパイラのアーキテクチャを提案し，この有用性について述べる．第 8 章では，IoT システムのアーキテクチャを提案し，この有用性について述べる．第 9 章では，本研究のアプローチおよび PBR パターンについての考察を述べる．第 10 章では，本研究の結論ついて述べて論文を締めくくる．

第 2 章

関連研究

本章では，自己適応ソフトウェアに関連する既存研究として，Salehie らの提案する自己適応ソフトウェアに関する既存の研究成果を分類する局面について述べる．開発を支援するために提案すされている関連技術として，代表的な概念モデルである MAPE-K，アーキテクチャスタイルである C2 スタイル，Weaves，アーキテクチャおよびその実現である CASA，K-Components，Rainbow について述べる．最後に，本研究の位置付けを述べる．

2.1 自己適応に関する既存の研究成果を分類する局面

Salehie ら [55] は，自己適応ソフトウェアの構造に着目し，既存の研究成果を 14 の局面で分類し，この局面を 4 つのグループにまとめている．以下にそれぞれのグループと局面を説明する．

適応対象

1. レベル：水平応用領域．アプリケーションソフトウェアやミドルウェア，基本ソフトウェアなど．
2. 粒度：コンポーネント，コンポーネント群，サブシステム等などの適応対象の粒度．
3. コスト：適応の実現および実行にかかるコスト．本研究では，統一的な単純な記述を目指すことから着目しない．

実現方法

4. 動的/静的：動的な織込み/静的な織込み．
5. 外部適応/内部適応：外部環境に応じた自己適応/システムの内部状態に応じた自己適応．コンテキストアウェアネスは外部適応に分類され，耐故障処理は内部適応に分類される．
6. ソフトコンピューティング/ハードコンピューティング：知的な適応処理を含むかどうかによって分類する局面．AI による自己適応はソフトコンピューティングに分類される．
7. オープン適応/クローズ適応：自己適応処理の追加変更の有無．追加変更がある自己適応はオープン適応に分類される．
8. モデルベース適応/モデルフリー適応：適応ポリシーを決定づけるモデルの有無．数理モデルに基づく自己適応はモデルベース適応に分類される．
9. 応用領域依存/一般化：垂直応用領域へ依存したものは応用領域特化に分類される．

時制特性

10. リアクティブ/プロアクティブ：自己適応を実行する時機．イベントが発生した後に適応を行なうものはリアクティブに分類され，発生を予想して適応を行なうものはプロアクティブに分類される．

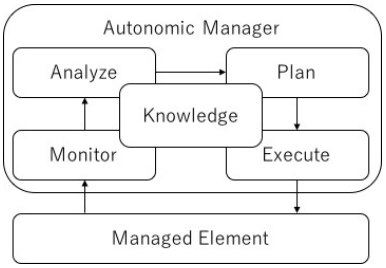


図 2.1: MAPE-K モデル [34]

11. 継続的モニタリング/適応的モニタリング：監視を行う時機．ポーリングによるイベント発生検知は継続的監視に分類され，インタラプトによるイベント発生検知は適応的監視に分類される．

適応結果の影響

12. 人間機械系：自己適応の結果が使用者インターフェースを介して使用者に影響を与えるかどうかで分類するための局面．
13. 適応結果の信頼性：機械的に自己適応の結果の良し悪しが判断できるものは信頼性が高いものに分類され，人間などの解釈が必要なものは信頼性が低いものに分類される．
14. 相互運用性：サブシステムの影響を受けて自己適応を行なうものかによって分類．

2.2 自己適応のための関連技術

2.2.1 MAPE-K

自己適応ソフトウェアの概念モデルとしては MAPE-K[34] がよく知られている．そのモデルを図 2.1 に示す．モデルは管理対象エレメント (*Managed Element*) と自律マネージャ (*Automic Manager*) に分割し，自律マネージャは，*Monitor*，*Analyze*，*Plan*，*Execute*，*Knowledge* の 5 つの要素から構成される．それぞれの役割は以下の通りである．

1. *Monitor*：環境を監視
2. *Analyze*：監視した情報を分析
3. *Plan*：Analyze の分析結果に基づいて再構成を計画
4. *Execute*：再構成を実行
5. *Knowledge*：自己適応の実行に必要な知識を保持

MAPE-K は，この 5 つのコンポーネントすべての動的な取り扱いを念頭に置いたものであり，ソフトウェアの動的進化を視野に入れている．

自己適応記述支援を目的としたアーキテクチャパターンとしては C2 コネクタ [63] と Weaves[30] がよく知られている．両者とも，並行オブジェクト指向計算モデルを前提としており，C2 は階層アーキテクチャなど構造があらかじめ定義されていることを想定している．一方，Weaves はあらかじめ想定される構造がないので，設計の自由度は高い．

2.2.2 C2 スタイル

C2 スタイルは，コンポーネントベースおよびメッセージベースのアーキテクチャスタイルである．コンポーネントとコネクタから構成される．それぞれのコンポーネントは並行に動作し，コネクタに接続される．

コンポーネント間の通信は，コネクタを介したメッセージ交換によって行われる．コンポーネントは上位のコンポーネントのみ依存して実現され，下位のコンポーネントに対して独立した階層構造として実現される．コネクタは，環境に応じてコンポーネント間のメッセージの経路を変更することにより，コンポーネント間の関係を再構成する．

図 2.2 は C2 スタイルのコンポーネントの内部アーキテクチャである．ダイアログは要求と通知のメッセージを受け取り，対応する内部オブジェクトの処理を実行する．内部オブジェクトは，状態が変化すると，このことを通知するメッセージを下位コンポーネントにブロードキャストする．下位コンポーネントのダイアログは，これを受け取り，適切な内部オブジェクトの処理を実行する．ドメイントランスレータは，通信の非互換性を解消するために，メッセージ形式を変換する．また，ドメイントランスレータを介して通信を行なうことでコンポーネントの独立性を確保する．

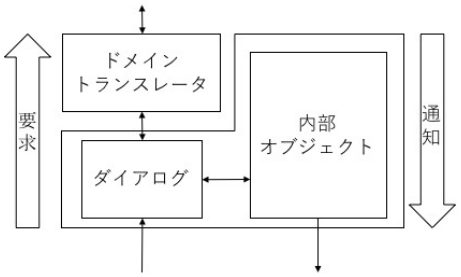


図 2.2: C2 スタイルのコンポーネントの内部アーキテクチャ

2.2.3 Weaves

Weaves は，動的再構成を重点に置いたデータフロースタイルである．データフローを処理するツールフラグメントの集合として定義される．各ツールフラグメントは並行に動作し，キューに接続されたポートを介してデータの送受信を行なう．ポートは，ツールフラグメントをキューに接続し，ツールフラグメント間の通信をバッファして同期させる．ポートおよびキューは受動的に動作し，ツールフラグメントは能動的に動作する．ツールフラグメントはデータフローを監視し，動的にデータを送信するポートを変更することにより，動的再構成を実現する．図 2.3 は Weves のアーキテクチャである．ツールフラグメントは，読み・書きの二種類のポートと接続され，このポートはキューと接続される．ポートはメッセージとして送られるデータオブジェクトをエンベロープでラッピング・アンラッピングを行ない，通信を実現する．

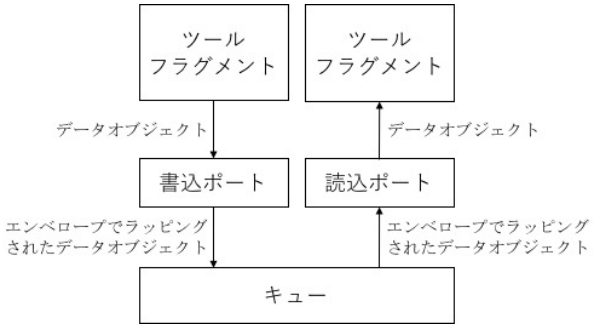


図 2.3: Weaves アーキテクチャスタイル

2.2.4 CASA(Contract-based Adaptive Software Architecuture)

CASA[46] は，様々な自己適応を実現するためのアーキテクチャおよびその実現としてのアプリケーションフレームワークを定義している．変化する環境に対して継続して要求を充足するために，リソースの消費方法の変更などの小規模な自己適応だけでなく，アプリケーションコードを変更する大規模な自己適応を実現することを課題としている．この課題を解決するために，アーキテクチャでは，次の 4 つを自己適応の対象としている．

1. 属性
2. コンポーネント
3. アスペクト
4. ミドルウェア

適応方法として，次の 2 つに対応している．

1. 緩やかな再構成 (Lazy Replacement)
2. 懸命な再構成 (Eager Replacement)

緩やかな再構成は，現在の処理が完了してから再構成を実行する．懸命な再構成は，現在の処理を中断して再構成を実行する．この CASA アーキテクチャを図 2.4 に示す．

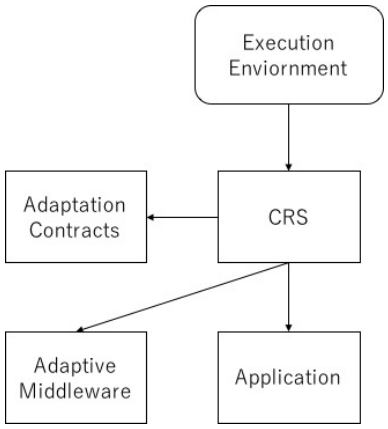


図 2.4: CASA アーキテクチャ

CASA は，Adaptation Contract，Adaptive Middleware，Application によって構成される．Adaptation Contract は，条件と条件に応じた構成が記述される．Adaptive Middleware は，自己適応を実行する処理系であり，Application を変更する．

2.2.5 K-Component

K-Components[20] は，分散システムのパフォーマンスの維持および最適化，障害からの復旧のための自己適応アーキテクチャである．分散システムでは，システム全体の状態に応じた再構成を実現するには，通信のオーバーヘッドがあることから，隣接するコンポーネントの状態からシステム全体の状態を推論し自己適応を実現することを課題としている．

変化する環境に対して適応動作を行なうコンポーネントおよび強化学習による協調の調整により，システム全体の品質を維持する．コンポーネントとコネクタの状態をフィードバックし，適応条件が満たされたさいに

適応アクションを実行する．また，このコンポーネント間ではフィードバックイベントによって，フィードバック情報を伝達する．K-Components アーキテクチャを図 2.5 に示す．

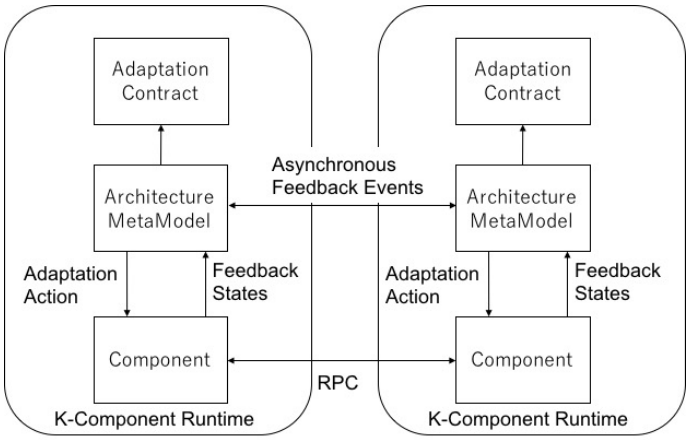


図 2.5: K-Components アーキテクチャ

K-Component は，Adaptation Contract，Architecture MetaModel，Component によって構成される．Architecture MetaModel は，Component のメタ記述である．Adaptation Contract は，条件と条件に応じた構成が記述される．Adaptation Contract に基づいて，特定の条件のとき，Architecture MetaModel を変更し，モデルにこの変更を反映させる．

2.2.6 Rainbow

ユーザのニーズ，リソース，障害状況などの変化に対応するための動的再構成のアーキテクチャとして，Rainbow[12, 29] が提案されている．Rainbow は，多様なソフトウェアに対する自己適応の適用を可能とすること，および，自己適応に関する処理を追加するさいのコストを削減することを課題としている．多様なソフトウェアに対する自己適応性の適用を可能とするために，関心のある特性および適応ポリシーを対象システムに応じて調整可能なインフラストラクチャを定義している．このインフラストラクチャは，アーキテクチャモデルの変更やモニタリング等を実現するために再利用可能な汎用的なものとして定義している．

Rainbow は，外部からシステムを監視し，システム API を用いて対象システムを構成を動的に変更する．Rainbow アーキテクチャを図 2.6 に示す．

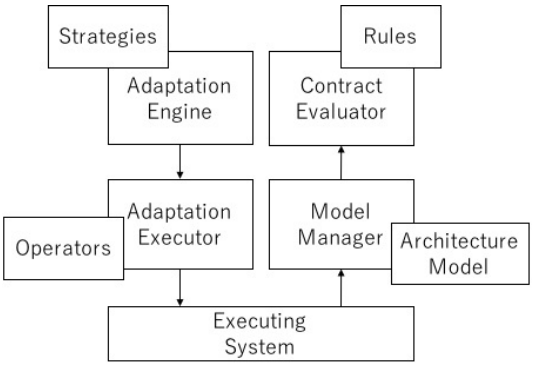


図 2.6: Rainbow アーキテクチャ

Rainbow は，Architectural Model，Rules，Strategy によって構成される．Architectural Model は，再構成対象のメタモデルであり，コンテキスト情報をプロパティとして持つ．Rules には，再構成を行なう条件と

なる Architectural Model のプロパティとその時に選択される Strategy の関係が記述される．Strategy には，Operator を用いた再構成のための Architectural Model のプロパティの変更処理が記述される．Operator には，Architectural Model に対して構成を変更を行なうために実行可能なアクションが定義される．

2.3 本研究の位置付け

関連研究では，特定の自己適応計算方法に依存したアーキテクチャおよびその実現が定義されている．C2 および Weaves は，階層間の通信における自己適応およびオブジェクト間の通信における自己適応を実現する．

特定の自己適応計算方法および応用領域に依存したアプリケーションフレームワークとして，CASA，K-Components，Rainbow が提案されている．一般的な実用システム毎に様々な自己適応が求められ，この要求に応じたアーキテクチャやその実現を選択して開発が行われる．例えば，K-Components は独立したコンポーネント毎の自己適応や，学習アルゴリズムによる自己適応を可能としている．CASA は，複数種類の適応方法に対応している．Rainbow は外部からアプリケーションを監視することにより，障害を検知し，自己回復を可能としている．

本研究では，Salehie らの提案する局面の値および応用領域によらず自己適応計算を取り扱う枠組みを提案することにより，自己適応ソフトウェアの作成を支援する．次章では，本研究における技術課題およびアプローチについて述べる．

第 3 章

技術課題と研究の進め方

本章では，自己適応ソフトウェアの作成支援を実現するための技術課題および研究の進め方について述べる．

3.1 技術課題

本研究の技術課題は，“Salehie らに分類される自己適応計算方法それぞれのためのパターンを導出できる共通のメタパターンを定義すること”として目的から具体化した．本研究は，構造だけでなくソフトウェアプロセス定義や PLSE(Product Line Software Engineering) 応用についても視野にいれ，アーキテクチャ中心の自己適応ソフトウェアの開発支援について考察する．多様な自己適応計算方法の振舞いを “イベントの検知–活性化–操作”として一般化し，総称型としてのメタパターンが定義できると考えた．メタパターンから特定の自己適応計算方法のためのアーキテクチャパターン，および，デザインパターン，コーディングパターンを導出し，多様な自己適応計算方法を統一的に扱う．一般に特定の計算方法や応用領域に依存して実現される自己適応ソフトウェアの開発に対して，単一のメタパターンから，様々な自己適応計算を記述し，統一的なソフトウェアプロセスによる開発を可能にする．以下の 4 つの応用領域の事例では，この分類される自己適応計算方法をすべて扱う．メタパターンから導出されるベースパターンを適用し，それぞれの自己適応に関連する技術課題を解決する．

組込みシステムにおいては，次の２つを技術課題とする．

1. 組込みシステムに要求される非機能特性を横断的コンサーンとして，独立してモジュール化する
2. 自己適応の問題としてコンテキスト指向およびアスペクト指向を再定義し，統一的に扱う

前述のように特定の非機能特性の分離や特定のコンテキストに依存したコンテキスト指向計算の実現に関する試みがなされている．自己適応の問題として再定義することで，コンテキスト指向およびアスペクト指向を両立し，単純な構造で記述可能になると考えた．コンテキスト指向およびアスペクト指向は，自己適応として位置付けることができるとの考えから，前述の目的をこれらの技術課題に具体化した．自己適応では，自己表現 (Self Representation) と振舞いが因果的結合 (Causally Connected) の関係にある．コンテキスト指向はコンテキストを自己表現とし，アスペクト指向は合流点を自己表現とした自己適応として位置付けることができる．単一の自己適応のためのメタパターンを用いて，コンテキスト指向およびアスペクト指向を実現する．また，実時間制約を考慮して自己適応の実現方法を選択し，実現するための枠組みを定義する．

インタラクティブシステムにおいては，次の２つを技術課題とする．

1. MVC やその他の派生アーキテクチャスタイルに基づく参照アーキテクチャに共通なメタ参照アーキテクチャを設計する
2. 使用者インターフェースの動的変更の実現をソフトコンピューティングによる自己適応の適応ポリシーの問題として解決する

アーキテクチャスタイルと実現技術の間には複雑な依存関係が存在するので，特定のアーキテクチャスタイル

や実現技術を，異なる技術に転換することは一般に容易ではない．メタ参照アーキテクチャを介して実現技術間の関係が整理され，実現技術の置換や再利用による開発を支援可能になると考えた．MVC やその他の派生アーキテクチャスタイルはそれぞれ異なる横断的コンサーンの分離を試みているとの考えから，目的を 1. の技術課題に具体化した．メタ参照アーキテクチャは，この横断的コンサーンによって規定されるアスペクトを統合するものとして定義する．アスペクトを織込むことで，特定のアーキテクチャスタイルに基づく参照アーキテクチャを導出する．ハードコンピューティングおよびソフトコンピューティングによる自己適応は，適応ポリシーの記述方法の問題であるとの考えから，目的を 2. の技術課題に具体化した．メタパターンの適応ポリシーの具象化により，ハードコンピューティングおよびソフトコンピューティングを実現可能にする．

モデルコンパイラにおいては，次の２つを技術課題とする．

1. 多様なモデルコンパイラを統一的に扱うアーキテクチャを設計する
2. モデルコンパイラの対象モデルの変更をメタモデルコンパイラによる適応処理によって実現する

前述のようにモデルコンパイラはその変換方法や対象モデルに応じて個別に開発が行われる．モデルコンパイラは変換方法により分類されると考えから，目的を 1. の技術課題に具体化した．入出力形式によって定義される横断的コンサーンを提案パターンにより分離し，モデルコンパイラの統一アーキテクチャを設計する．横断的コンサーンを指定し，アスペクトを織込むことで，特定のモデルコンパイラのアーキテクチャを導出する．提案アーキテクチャにより多様なモデルコンパイラは統一的に説明され，そのモジュールに既存技術が対応付けられる．これにより，モデルコンパイラの既存技術の柔軟な選択および統合による開発を支援できると考えた．言語モデルをパラメータとしたメタモデルコンパイラのモデルフリー適応によって，モデルコンパイラの対象モデルの柔軟な変更が可能になるとの考えから，目的を 2. の技術課題に具体化した．モデルおよび変換規則を入力としたメタモデルコンパイラによってモデルコンパイラの対象モデルを変更する．

IoT システムにおいては，コンテキスト協調は他のソフトウェアの影響を起因とした外部適応という考えから，メタレベルにおける外部適応記述のための単純な構造を定義することを技術課題とした．メタレベルの自己適応記述は，スクリプトの追加等により実行時に変更可能なものとすれば，自己適応サブシステム間の協調が動的に変更可能なオープン適応の実現として考察できる．自己適応のためのアーキテクチャを自己反映的に適用することでコンテキスト協調のための単純な構造を得る．コンテキスト協調について安直に記述した場合と比較し，単純な記述が得られると考えた．

3.2 研究の進め方

Salehie らの局面は完全独立とみなし，各局面における値を網羅する前述の 4 つの事例に対して，メタパターンから導出されたパターンを適用する．この結果を一般化することで Salehie らの分類するすべての自己適応計算が記述できることを確認する．Salehie らの提案する局面と前述の 4 つの事例の関係は次の 2 種類に分類できる．

1. 特定の事例で局面の値すべてを扱うもの
2. 複数の事例で局面の値すべてを扱うもの

1. については，特定の事例に提案パターンを用いて，局面の値すべてを扱い可能であることを確認する．例えば，組込みシステムの事例では，動的/静的適応局面の値である動的適応と静的適応を両方とも扱うことから，提案パターンを用いて，これらを統一的に取り扱えることを考察する．2. については，複数の事例に提案パターンを適用し，局面の値すべてを扱い可能であることを確認する．例えば，レベルすなわち水平応用領域によって分類する局面については，それぞれの水平応用領域の事例に対して，提案パターンを適用し，複数事例の結果をまとめることで，統一的に取り扱えることを考察する．

アーキテクチャパターンを導出・適用することにより，参照アーキテクチャが設計できることを確認する．メタパターンのコンポーネントに，アーキテクチャレベルの自己適応計算の役割を与え，特定の自己適応計算

方法が記述できることを確認する．

参照アーキテクチャに基づき，デザインパターンを導出・適用することで，具象アーキテクチャが設計できることを確認する．参照アーキテクチャのコンポーネントに対し，デザインレベルの自己適応計算の役割を与え，詳細化することで設計する．デザインパターンに定義されるコーディングパターンを導出・適用することにより，特定のプログラミング言語を用いたコードの記述方式を定義する．

第 4 章

PBR パターン

ハードウェアのダウンサイジングならびにギガビットイーサをはじめとする高速通信技術の地球規模での普及に伴い，自己適応性が従来にも増してソフトウェアに求められるようになってきた．モバイル計算の実用化に伴い，インタラクティブシステムおよび組込みシステムは，移動体として実現されることが多い．これらは，取り巻く外部環境に応じて自己適応的に振舞いを変化させる．IoT システムでは，組込みシステムの一部をサービスとして実現する．そのサービスの振舞いも，協調する移動体に応じてその振舞いを変化させる．自己適応のためのパターンとして，PBR(Policy-Based Reconfiguration) パターンを定義し，自己適応ソフトウェアアーキテクチャの設計および実装を支援する．

4.1 PBR パターンの設計

異なる自己適応計算方法を統一的に取り扱うために，共通なメタパターンとして PBR パターンを定義した．このメタパターンから特定の計算方法に依存した (ベース) パターンを導出する．本研究においては，参照アーキテクチャ設計，具象アーキテクチャ設計，コーディングに PBR パターンを用いることで，自己適応計算を統一的に取り扱う．

4.1.1 PBR パターン

メタパターンに対して目的 (インテント) を与えることにより，(ベース) パターンを得る．この目的 (インテント) は，メタパターンのコンポーネントそれぞれにその役割を与えることによって指定する．役割を与えることによって具象化する総称型としてメタパターンを定義した．

PBR パターンの構造は，最も広く使用されているアスペクト指向プログラミング言語である AspectJ のアドバイス記述を一般化して設計した．これまでに記述されてきた AspectJ コードを観察した結果，アドバイス記述には，合流点によってイベントを検知し，このイベントに応じたアスペクトモジュールの構成と起動に関連する記述がなされることを確認した．このアドバイス記述の振舞いを抽象化し，“イベントの検知–活性化–操作”の構造として素直にモデル化した．

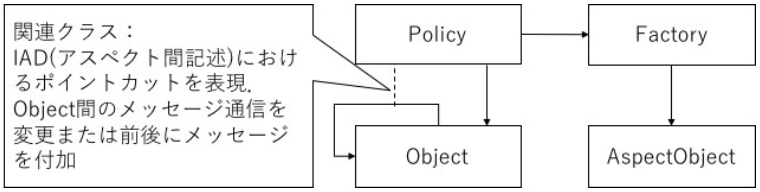
以下に，PBR パターンカタログを示す．

[**名前**] PBR パターン

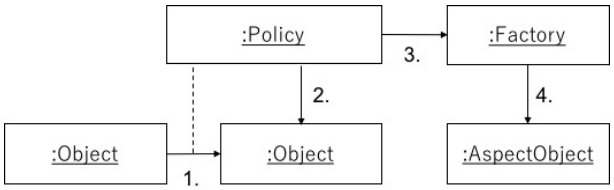
[**目的**] 異なる自己適応計算を統一的に取り扱う．PBR パターンのコンポーネントに役割を与えることで，特定の自己適応計算方法のためのパターンを導出する

[**課題**] 自己適応は再構成に還元できる．すなわち，特定のポリシー (コンテキストの変化を含む) に応じて，ソフトウェアの構成を変更することを指す．このさい，特定のポリシーと再構成の仕組みを独立に記述するというフォースを考慮しなければならない．

[**解決策**] ポリシーと再構成の仕組みをそれぞれコンポーネント化し，実現する．PBR パターンの静的構造と動的振舞いを図 4.1(a)，(b) に示す．コンテキストの変化を含むポリシー (*Policy*)，ポリシーに応じて変化す



(a) 静的構造



(b) 動的振舞い

図 4.1: PBR パターン

表 4.1: MAPE-K と PBR パターンとの関係

MAPE-K	PBR パターン
Monitor	IAD(アスペクト間記述) におけるポイントカット
Analyze	Policy
Plan	Policy
Execute	Factory

る再構成後のオブジェクト群を代表するアスペクトオブジェクト (*AspectObject*)，再構成の仕組みとして，このアスペクトオブジェクト (*AspectObject*) のインスタンス生成を行なうファクトリ (*Factory*) から構成した．ポリシー (*Policy*) がその記述に従ってファクトリ (*Factory*) を起動する関係となり，ファクトリ (*Factory*) はアスペクトオブジェクト (*AspectObject*) を生成する関係となる．*Object* 間のメッセージを横取りし，*Policy* で，*Factory* に *AspectObject* のインスタンスを生成させ，このインスタンスにメッセージを送る．

アーキテクチャレベルからコードレベルまで設計支援を実現するために，メタパターンのコンポーネントそれぞれに対して役割を与えることで，アーキテクチャパターン，デザインパターン，コーディングパターンに具象化する．これらパターンを事例に適用し，参照アーキテクチャ設計，具象アーキテクチャ設計，コーディングを行なう．

4.1.2 PBR パターンと既存の自己適応技術との比較

PBR パターンは MAPE-K モデルを単純化して定義したものとも考えられる．その対応を表 4.2 に示す．*IAD*(アスペクト間記述) におけるポイントカットは，メッセージ通信から環境の変化を識別するものであり，*Monitor* としての役割を持つ．*Policy* は，識別された環境から再構成すべき状況かどうかを判断し，適切な構成を選択するものであり，*Analyze* および *Plan* としての役割を持つ．*Factory* は，分析された状況から再構成を行なうものであり，*Execute* としての役割を持つ．PBR パターンのポリシーとファクトリの記述において，すべて動的に取り扱えば，MAPE-K が本来念頭に置いている動的ソフトウェア進化に対応可能である．他方，すべて静的に取り扱えば，アスペクト織込みに相当する記述となる．

PBR パターンは C2 や Weaves を抽象化したものとも解釈できる．これは，アドバイス記述を複数のコン

表 4.2: 自己適応記述支援のためのアーキテクチャパターンと PBR パターンとの対応関係

(a) C2 と PBR パターン

C2	PBR パターン
C2 コネクタ内に定義される 環境に応じた再構成のポリシー	Policy
C2 コネクタ内に定義される 自己適応の手続き	Factory
異なる下位コンポーネントと 接続された C2 コネクタ	AspectObject

(b) Weaves と PBR パターン

Weaves	PBR パターン
ウィーブ内に定義される 環境に応じた再構成のポリシー	Policy
ウィーブ内に定義される のパイプ再構築の手続き	Factory
異なるオブジェクトと 接続されたパイプ	AspectObject

ポーネントに分割したことにより，それぞれのコンポーネントの役割の汎用性が高くなった結果と考える．その対応関係を表 4.2(a)，(b) に示す．C2 は，コンポーネントとこれを関連付けるコネクタによって構成される階層モデルである．コンポーネント間を関連付けるコネクタが自己適応的に再構成する．Weaves は，オブジェクトとこれを関連づけるパイプおよび，パイプ群を管理するウィーブによって構成される．ウィーブが自己適応的に再構成することで，パイプが変更され，オブジェクト間の関係が変化する．PBR パターンでは，このコネクタおよびパイプの変更を，これらのインスタンスの再生成によって対処する．C2 と Weaves におけるコンポーネントおよびパイプは *AspectObject*，コネクタおよびウィーブ内に定義される再構成のためのポリシーは *Policy*，コネクタおよびウィーブ内に定義されるコネクタおよびパイプのインスタンスの再生成の手続きは *Factory* に対応する．

4.2 まとめ

アーキテクチャレベルからコードレベルまで自己適応計算を統一的に扱うためのパターンとして PBR パターンを定義した．PBR パターンはメタパターンであり，この単一のメタパターンに対して目的 (インテント) を与えることにより，それぞれの抽象度における特定の自己適応計算方法のためのベースパターンを導出することができる．目的 (インテント) は，コンポーネントそれぞれに役割を与えることによって指定する．メタアーキテクチャパターンとしての PBR パターンからアーキテクチャパターンの導出する．メタデザインパターンに，既存のデザインパターンを与えることで，デザインパターンを導出する．メタコーディングパターンに，特定の言語の言語要素を与えることで，コーディングパターンの導出する．このように，ベースパターンを導出し，適用することで統一的な取り扱いを実現する．

以下の章では，複数の応用領域の事例の自己適応計算に関する技術課題の解決のために PBR パターンを用いる．それぞれの事例における PBR パターンの有用性についても考察する．

第 5 章

組込みシステムのためのソフトウェアアーキテクチャ

モバイル計算の実用化にともない，組込みシステムは移動体として設計・実現されることが多くなってきた．このような組込みシステムはそれを取り巻く環境を反映する内部状態をコンテキストとし，コンテキストに応じてその振舞いを変化させる．一方，組込みシステムでは並行性，実時間性，耐故障性などの非機能特性についても適切なモジュール化が重要となる．本論文では，コンテキストおよび横断的コンサーンとしての非機能特性を統一的に扱う組込みシステムのためのアーキテクチャを設計し，その有用性について議論する．PBR パターンを用いてコンテキストおよび非機能特性を統一的に扱うことを可能とした．アーキテクチャとコードの理解，変更が容易になるだけでなく，ライブラリ等を大きな粒度で再利用する枠組みが提供できた．

5.1 組込みシステムの開発の現状および問題点と解決策

組込みシステムは，並行に動作するハードウェアの集合によって実現される．近年，組込みシステムを，取り巻く外部環境に応じて自己適応的に設計・実現する試みが盛んに行われている [18]．すなわち，外部環境を反映するシステムの内部状態をコンテキストとし，コンテキストアウェアネス [55] 実現を試みている．モバイル計算が実用化され，組込みシステムを移動体として設計・実現する場合，この傾向はより顕著になる．加えて，組込みシステムの開発では，実時間性，耐故障性などの非機能特性も設計し，実現しなければならない [1, 32]．過去の開発経験から，組込みシステムにおいてこれらの非機能特性をセルフアウェアネス [55] 実現することが自然との認識に至った．以上をまとめると，オブジェクト指向を支配的分割 (以下，コアコンサーン) とし，コンテキストコンサーンおよび横断的コンサーンとして非機能特性を適切にモジュール化することが重要となる．

本研究は，アーキテクチャ中心開発の基盤となる組込みシステムのためのアーキテクチャを設計することを目的とする．この目的は，次の 2 つの技術課題に具体化できる．

1. 前述の横断的コンサーンを矛盾なくモジュール化する
2. 自己適応の問題としてコンテキスト指向およびアスペクト指向を再定義する

自己適応計算は，自己記述に基づいて振舞いを制御する．コンテキスト指向では，コンテキストを自己記述とした自己適応計算である．アスペクト指向は，合流点を自己記述とすれば，自己適応計算として位置付けることができる．すなわち，アドバイス記述により自己記述の活性化に応じて振舞いが制御される．我々はコンテキスト指向およびアスペクト指向を統一的に取り扱うために，自己適応のためのパターンとしての PBR パターンを用いる．

PBR パターンを事例に適用し，設計されたアーキテクチャにおけるコンテキストおよび横断的コンサーンの統一的な取り扱いとその利点について考察する．

本論文で提案するアーキテクチャは，実際の組込みシステム開発に適用し実用性を確認したアーキテクチャ

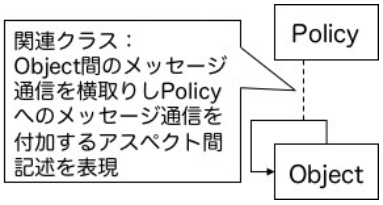


図 5.1: 本研究におけるアスペクト間記述の記法

[67] を改版したものである．PBR パターンを適用することで，アーキテクチャとアプリケーションの設計，およびコードの理解，変更が容易になるだけでなく，ライブラリ等を大きな粒度で再利用する枠組みが提供できた．

5.2 アーキテクチャ設計

OASIS のアーキテクチャの定義 [42] が一般的なアーキテクチャの設計・運用の枠組みを定義しているとの認識に立ち，これに基づいてアーキテクチャについて議論する．アーキテクチャ設計にあたり，横断的コンサーンを統一的に扱うために，自己適応のための PBR パターンを定義し，これを適用してアーキテクチャを設計する．

本研究では, アスペクト指向モデリングの記法として, Theme/UML[14] の略式記法を用いる．Theme/UML は UML を拡張して定義されたアスペクト指向モデリングのための一般的な記法である．本研究では，アスペクトの構造をクラス図とコラボレーション図を用いて表現し，Theme/UML において bind 関連線で表現されるアスペクト間記述を，関連クラスに置き換えてこれらの図内に表現する．例えば，図 5.1 は，*Object* 間のメッセージ通信に *Policy* へのメッセージ通信を付加するアスペクト間記述を関連クラスを用いて表現している．

5.2.1 組込みシステムの横断的コンサーン

組込みシステムは，物理的な対象を制御し，これらは並行に動作するので，一般にソフトウェア上で並行に動作するオブジェクトの集合として定義されている [40]．実時間性，耐故障性などの非機能特性も設計し，実現しなければならない [1, 32]．前述のように，移動体としての組込みシステムを考えた場合，コンテキストアウェアネス組込みシステムとして実現することは有用である [18]．これらをまとめると，オブジェクト指向をコアコンサーンとし，以下のコンサーンを適切にモジュール化することが重要である．

a) 並行性コンサーン
b) コンテキストコンサーン
c) 実時間性コンサーン
d) 耐故障性コンサーン

前述の横断的コンサーンおよびコンテキストコンサーンの分離を目的 (インテント) として，(ベース) パターンを導出する．PBR パターンのコンポーネントとベースパターン導出のために与える役割の関係を表 8.1 に示す．例えば，並行性コンサーンについては，図 5.2 に示すように，PBR パターンの *Policy*，*AspectObject*，*Factory* のそれぞれにに対して，スケジューリングのポリシーとして *SchedulingPolicy*，並行実体としてスレッド (*Thread*)，ポリシーに従ってスレッドの生成および活性化と非活性化を行なう *Scheduler* の役割を与えることで，並行性コンサーンを分離するためのベースパターンを得る．このベースパターンを適用し，コンポーネントの振舞いを具体化することでアーキテクチャを設計する．

表 5.1: PBR パターンのコンポーネントに与える役割

コンサーン	PBR パターンの コンポーネント	役割
並行性	Policy	SchedulingPolicy
	Factory	Scheduler
	Aspect Object	Thread
実時間性	Policy	TimingPolicy
	Factory	TimerFactory
	Aspect Object	Timer
耐故障性	Policy	Acceptance Policy
	Factory	F.T. HW Factory
	Aspect Object	HW
コンテキスト	Policy	Context
	Factory	Behavior Activator
	Aspect Object	Behavior

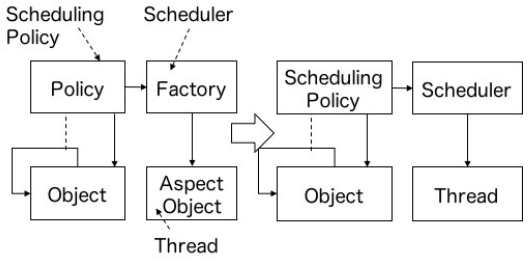


図 5.2: メタアーキテクチャパターンからアーキテクチャパターンへの具象化

5.2.2 参照モデル

山本は一般的な組込みシステムの参照モデルを示している [68]．本研究では，この参照モデルを組込みシステムの設計および実現の観点から抽象化した構造を定義した．本研究で定義した参照モデルを図 5.3 に示す．参照モデルの含意するところは，以下の通りである．

- 複数のセンサとアクチュエータが協調する．
- モニタにより，センサとアクチュエータは並行に動作する．
- 使用者はユーザインタフェースを介して，センサとアクチュエータに対する操作を行なうとともに，モ

ニタに対して並行実行処理等に関する操作を行なう．

図 5.3 では，隣接する階層間でメッセージの授受があることを示している．

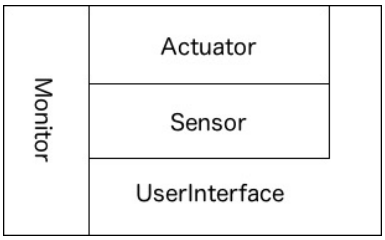


図 5.3: 組込みシステムのための参照モデル

5.2.3 参照アーキテクチャ

コンテキストおよび複数の非機能特性を矛盾なくモジュール化することを目的とし，具象化したパターンを適用することでアスペクト指向参照アーキテクチャを設計した．

以下，設計した参照アーキテクチャについて説明する．

参照アーキテクチャの概要

図 5.4 に参照アーキテクチャの概要を示す．並行性コンサーンは組込みシステム全体に横断し，実時間性コンサーンおよび耐故障性コンサーンは，一部に横断する．コンテキストコンサーンは複数のコンポーネントそれぞれに関連する．

以下，PBR パターンに目的 (インテント) を与えて得られた (ベース) パターンを適用することで，非機能特性およびコンテキストを矛盾なくモジュール化した参照アーキテクチャを示す．

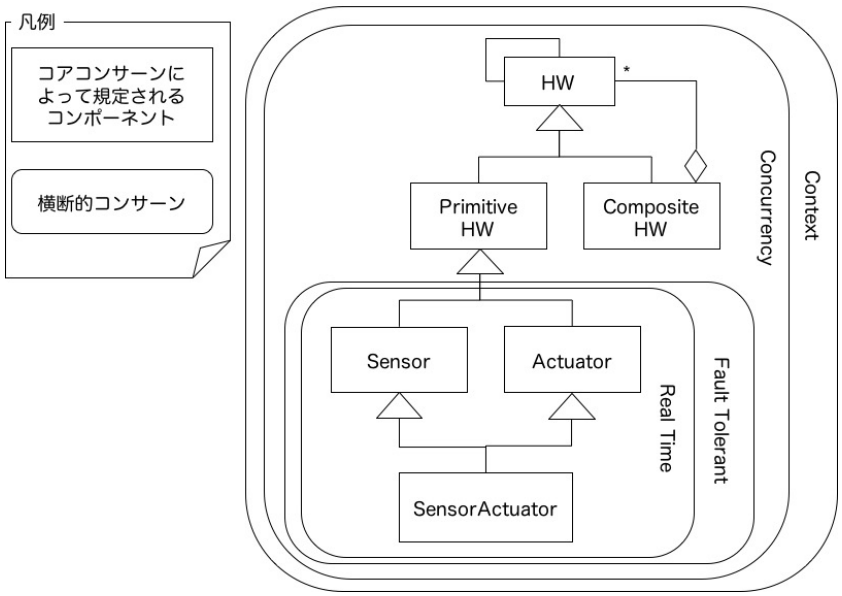
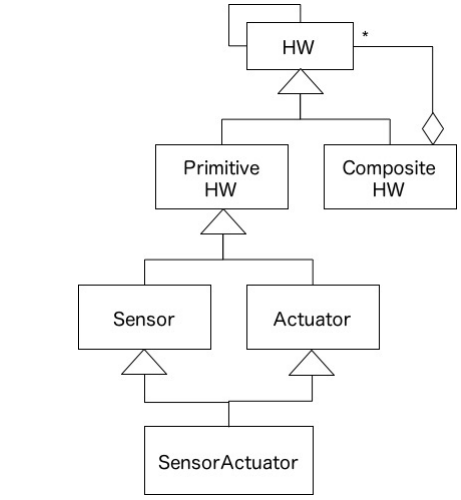


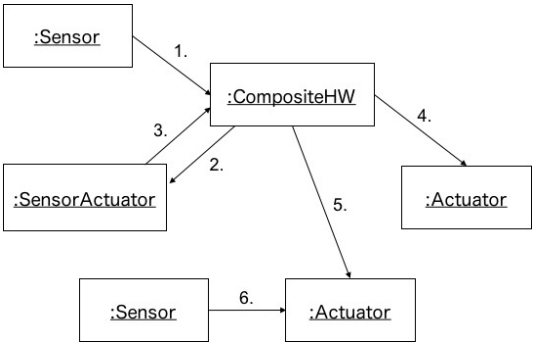
図 5.4: コアコンサーンと横断的コンサーンとの関係の概略

オブジェクト指向 (コアコンサーン)

組込みシステムでは，並行に動作する物理的なハードウェアに対してオブジェクトを定義する．このハードウェア (*HW*) の集合として設計した静的構造と動的振舞いを図 5.5(a)，(b) に示す．ハードウェア (*HW*) は，参照モデルに示されたようにセンサ (*Sensor*) とアクチュエータ (*Actuator*) に分類されるので，これらを多相型として定義した．センサ (*Sensor*) とアクチュエータ (*Actuator*) 両方の性質を持つもの (*SensorActuator*) に対しては多重 is-a 関係を用いて定義した．これら原始ハードウェア (*Primitive HW*) と複数の原始ハードウェアから構成される複合ハードウェア (*Composite HW*) を多相型として定義した．



(a) 静的構造



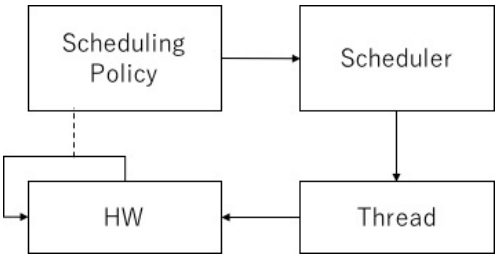
(b) 動的振舞い

図 5.5: ハードウェア

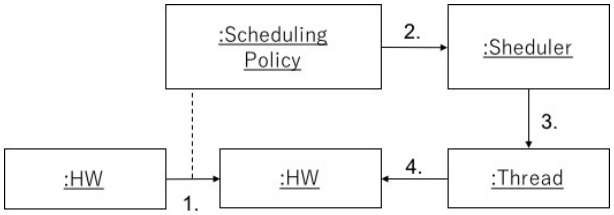
並行性

並並行プロセス間の同期に関わる記述を分離する．並行処理は，オブジェクト指向との親和性から，バッファ付きの非同期通信チャネル方式で実現することとした．並行性コンサーンを分離することで，コアコンサーンでは同期に関わる記述を考慮しなくて良くなる．PBR パターンから導出されるパターンを適用することで，スケジューリングのポリシーを独立に変更できるよう設計される．この静的構造と動的振舞いを図 5.6(a)，(b) に示す．並行性アスペクトをスケジューリングポリシー (*SchedulingPolicy*)，並行実体としてのスレッド (*Thread*)，スレッドの生成および活性化と非活性化を行なうスケジューラ (*Scheduler*) から構成した．*HW* 間のメッセージ通信を横取りし，*SchedulingPolicy* 内でメッセージをバッファに格納する．*Scheduler* は，*SchedulingPolicy* に定義されるポリシーに従って *Thread* の生成および活性化と非活性化を行なう．この

Thread は，活性化されたらバッファからメッセージを取得し，処理する．



(a) 静的構造



(b) 動的振舞い

図 5.6: 並行性

実時間性

実時間制約に関連する記述を分離する．このさい以下の 2 つの方式が考えられる．

1. タイマを用いて局所的に実現する．
2. 大域的なモニタを用意しそこの中で実現する．

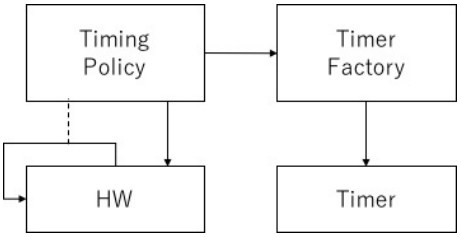
モニタによる実現は，計算モデルに関連する実現方法である．オブジェクト指向と親和性が高いタイマを用いる方法を採用することとした．PBR パターンから導出されるパターンを適用することで，実時間制約を実現するポリシーを独立に変更できるよう設計される．この静的構造と動的振舞いを図 5.7(a)，(b) に示す．実時間アスペクトを，実時間処理のポリシー (*TimingPolicy*)，タイマファクトリ (*TimerFactory*)，タイマ (*Timer*) から構成した．*HW* 間のメッセージ通信を横取りし，*RealTimePolicy* に従って，*TimerFactory* により *Timer* のインスタンスの生成，*Timer* の起動を行なう．

耐故障性

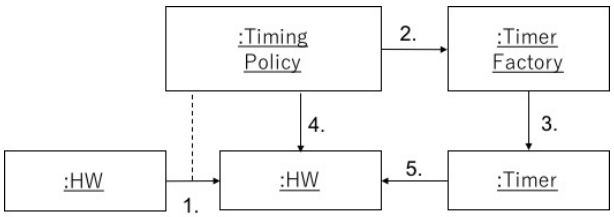
耐故障処理に関連する記述を分離する．一般に耐故障処理は，判定器によってバリアントを実行する．PBR パターンから導出されるパターンを適用することで，実行結果の受け入れ基準やバリアントを独立して変更できるように設計される．この静的構造と動的振舞いを図 5.8(a)，(b) に示す．耐故障アスペクトを，耐故障処理のポリシ (*AcceptancePolicy*)，耐故障ハードウェアファクトリ (*F.T.HWFactory*)，ハードウェア (*HW*) から構成した．*HW* 間のメッセージ通信を横取りし，*AcceptancePolicy* は，*HW* の処理結果を受理可能かどうか判断し，受理できない場合は代替ハードウェア (*HW*) にメッセージを送る．

コンテキスト

コンテキストに関連する記述を分離する．コンテキスト指向プログラミング言語にあるように，コンテキストとこれに応じた振舞い，この振舞いを活性化する手続きを分離し，独立に変更できるように設計される．この静的構造と動的振舞いを図 5.9(a)，(b) に示す．PBR パターンを適用し，ポリシーをコンテキスト (*Context*)，ファクトリを振舞い活性化手続き (*BehaviorActivator*) とした．*HW* 間のメッセージ通信を横取

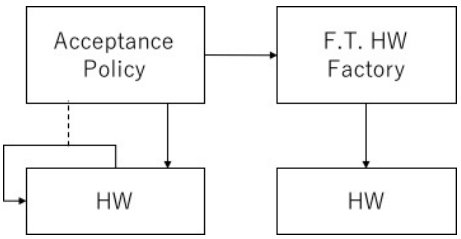


(a) 静的構造

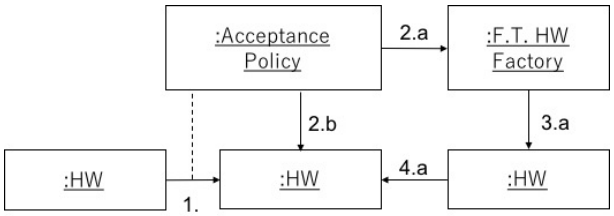


(b) 動的振舞い

図 5.7: 実時間ハードウェア



(a) 静的構造



(b) 動的振舞い

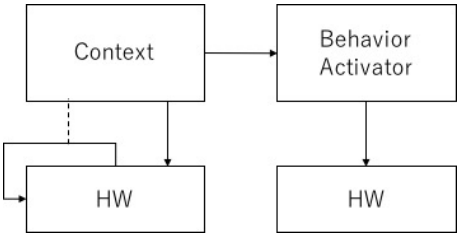
図 5.8: 耐故障ハードウェア

りし，*Context* の状態を変化させる．

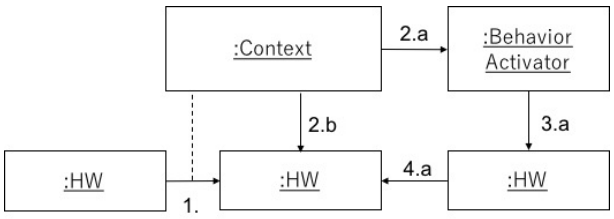
以上をまとめると，PBR パターンによって導出されるパターンにより，統一的に非機能特性およびコンテキストを矛盾なくモジュール化して記述できた．

5.2.4 具象アーキテクチャ

具象アーキテクチャは，実現技術を選択し，より具体的な役割をコンポーネントに与えることで，参照アーキテクチャの構造を詳細化したものである [42]．実現技術とは，製品特有のモジュール構成法，プロトコル，コード記述方法を指す．我々が過去に開発した紙状のものを搬送・管理するシステム (以下，紙状物体搬送システム) は，非機能特性およびコンテキストを扱う組込みシステムであることから，これを例題として具象アー



(a) 静的構造



(b) 動的振舞い

図 5.9: コンテキストアウェアハードウェア

キテクチャを説明する．

組込みシステムに適用される実現技術

対象とする組込みシステムにおける実現技術を整理すると以下の通りである．

- **モジュール構成法：**
  ハードウェアの構成法と，自己適応に関連する構成法がある．ハードウェアは一般に状態遷移機械としてモデル化される [27]．自己適用技術に関連する構成法は，C2[63]，Weaves[30] がある．層モデルなど構造が明確なものを指向する場合は C2[63] を，オブジェクト指向スタイル [58] など構造の自由度が高い場合は Weaves[30] を用いるべきであることが知られている．
- **プロトコル：**
  アプリケーションフレームワークやライブラリが提供されているとして，これらのアクセス方法が分かれば，アプリケーションを実装することができる．これをプロトコルの問題とする．並行プロセスの同期のためのプロトコルの代表的なものとして，Signal-Wait や PV 命令が選択可能である．耐故障処理のためのプロトコルの代表的なものとして，リカバリブロック，N バージョン，自己検査 (Self Checking Programming) が選択可能である [1]．コンポーネント間のメッセージ通信のプロトコルの代表的なものとして，リアクティブ (push 型) とポーリング (pull 型) が選択可能である．
- **コード記述方法：**
  アスペクトの記述方法の代表的なものとして，Java を基本とした AspectJ がある．

紙状物体搬送システムは，ハードウェアの構成が固定である．したがって，自己適用に関連するモジュール構成法として，C2 を選択した．前述のとおり，ハードウェアは，一般に状態遷移機械としてモデル化されることから，これに従った．

実現技術は従属的である．1 つの実現技術の選択により，別の実現技術の選択が決定する．例えば，プログラミング言語として，Java を選んだ場合，通常 Thread クラスライブラリを用いて並行処理を実現する．この Thread クラスを用いれば，必然的に同期のためのプロトコルは Signal-Wait となる．Java を用いて実現すること念頭に置いているので，Signal-Wait を選択した．状態遷移機械と従属的なメッセージ通信のプロトコル

として，リアクティブ (push 通信) を選択した．

組込みシステムではメモリ制約があり，プログラムの冗長性を許容することができない．したがって，実行時のコードサイズが小さくなる実現技術を選択しなければならない．このことから，リカバリブロックを選択した．

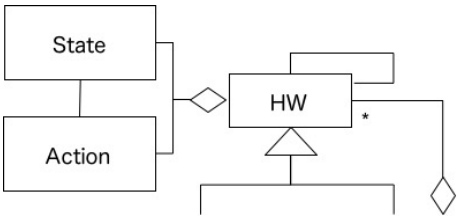
以上をまとめると，次の実現技術を選択した．

- **プログラミング言語：**Java (AspectJ)
- **自己適用技術に関連する構成法：**C2
- **モジュールの構成法：**ハードウェアを状態遷移機械としてモデル化
- **並行性に関するプロトコル：**Signal-Wait
- **耐故障性に関するプロトコル：**リカバリブロック
- **アスペクト記述法：**AspectJ
- **コンポーネント間通信：**リアクティブ

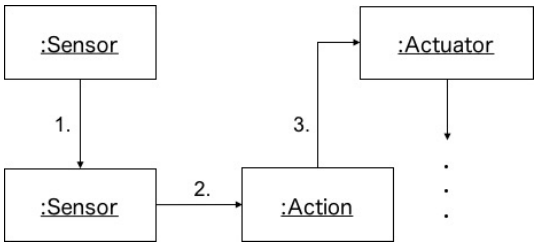
以下，これらの実現技術を前提とした具象アーキテクチャを設計する．

オブジェクト指向 (コアコンサーン)

ソフトウェアの保守性を考慮し，ミーリ型状態遷移機械を導入する．すなわち，現在の状態に応じたイベントとアクションの対を定義する．状態およびアクションや状態遷移の変更を独立して行なうことを目的として設計する．静的構造と動的振舞いを図 5.10(a)，(b) に示す．状態とアクションを別のモジュールとして定義した (*State*，*Action*)．これによりそれぞれのモジュールの独立性を確保する．多相型については，図 5.5 と同じなのでここでは省略する．状態 (*State*) はイベントに応じてアクション (*Action*) を実行し，遷移後の状態 (*State*) を返すことで状態遷移を表現する．



(a) 静的構造



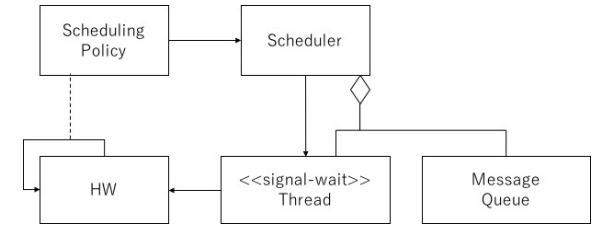
(b) 動的振舞い

図 5.10: ハードウェア

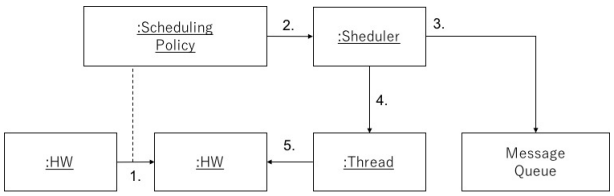
並行性

並行性コンサーンに関する静的構造と動的振舞いは，並行処理のためのライブラリの差替えを独立して行なうことを目的としてコンポーネントを詳細化する．この静的構造と動的振舞いを図 5.11(a)，(b) に示す．複数の並行に動作するオブジェクトからのメッセージを順に処理するために，このメッセージを管理するものと

して MessageQueue を導入した．各オブジェクトをメッセージキューでラッピングすることにより，Java の Thread クラスライブラリをそのままコンポーネントとして再利用可能とした．



(a) 静的構造



(b) 動的振舞い

図 5.11: 並行性

実時間性

実時間性コンサーンに関する静的構造と動的振舞いは，実時間制約の変更や実時間計測のためのライブラリの再利用を独立して行なうことを目的としてコンポーネントを詳細化する．静的構造と動的振舞いを図 5.12(a)，(b) に示す．ハードウェア (*HW*) に応じたタイマ (*Timer*) を起動するために，*TimerFactory* は *HW* と *Timer* の組を持つように設計した (図 5.12(a) 中の包含関係)．*TimingPolicy* は，計測時間や時間切れ時のアクションおよび *HW* と *Timer* の関係をパラメータとして *TimerFactory* に与えてインスタンスを生成することで，*TimerFactory* と *Timer* の再利用が可能となった．

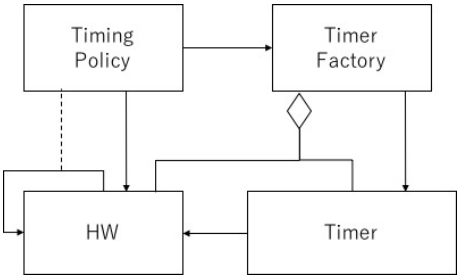
耐故障性

耐故障性プロトコルを独立して変更可能とすることを目的としてコンポーネントを詳細化する耐故障性プロトコルを *AcceptancePolicy* の内で実現するものとして定義した．静的構造と動的振舞いを図 5.13(a)，(b) に示す．*AcceptancePolicy* の内でプロトコルを実現していることから，この差し替えによりプロトコルの変更を容易に行なうことができ，バリアントはそのまま再利用することが可能である．

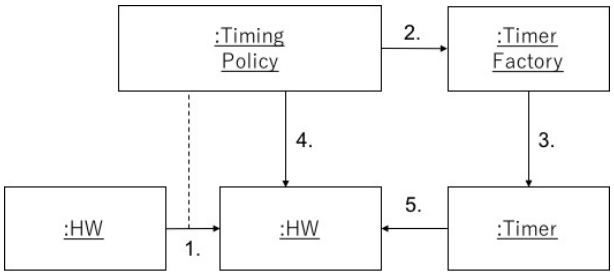
コンテキスト

コンテキストコンサーンに関する静的構造と動的振舞いは，コンテキストに依存して変化する差分のみを定義するようにすることを目的として設計する．静的構造と動的振舞いを図 5.14(a)，(b) に示す．ハードウェア (*HW*) を状態遷移機械として実現することから，この振舞いをコンテキストに応じて変更するために，コンテキストに依存したアクションの集合 (*BehaviorSet*) を持つ状態遷移機械ファミリ (*STMFamily*) を定義した．*Context* の状態に応じて *BehaviorActivator* は，*BehaviorSet* の持つ *Action* を組み合わせて *HW* を構築させる．これにより，差分となるアクションのみを独立して定義し，この組み合わせによってコンテキストに応じた構成を定義できるようになった．

以上を適用した紙幣搬送システムの具象アーキテクチャを図 5.15(a)，図 5.15(b) に示す．

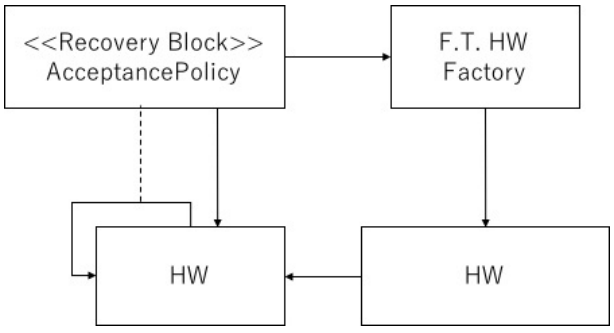


(a) 静的構造

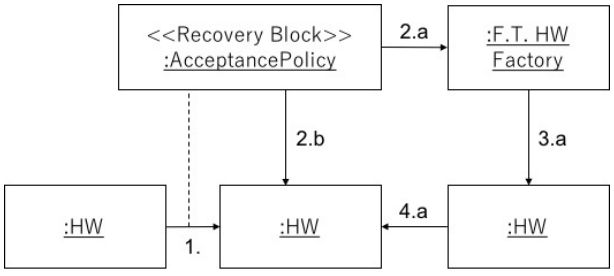


(b) 動的振舞い

図 5.12: 実時間ハードウェア

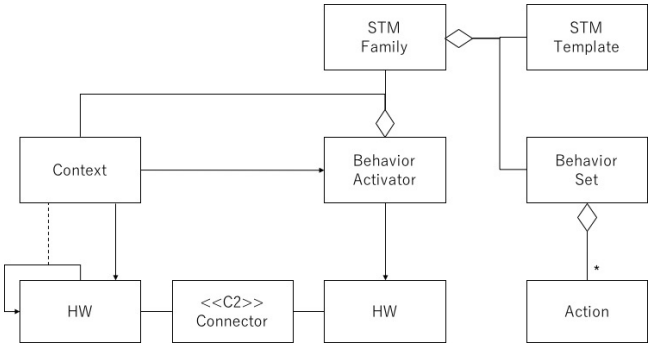


(a) 静的構造

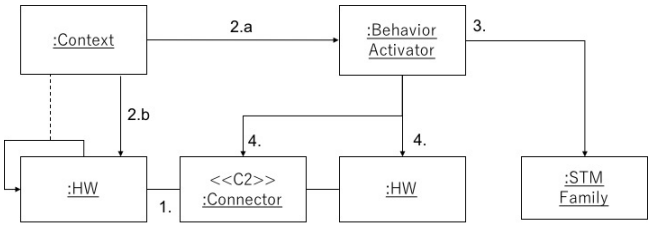


(b) 動的振舞い

図 5.13: 耐故障ハードウェア



(a) 静的構造



(b) 動的振舞い

図 5.14: コンテキストアウェアハードウェア

5.3 考察

本研究で調査した範囲では，コンテキスト指向技術を組込みシステムの設計・実装に応用する研究は行われているが [18]，アスペクト指向とコンテキスト指向を統一的に扱う試みは行われていない．本研究では，自己適応のためのメタパターンとしての PBR パターンを用いることで，これらを統一的に取り扱いとなることを確認した．この PBR パターンを用いることによる利点として次の 3 つの事項が挙げられる．

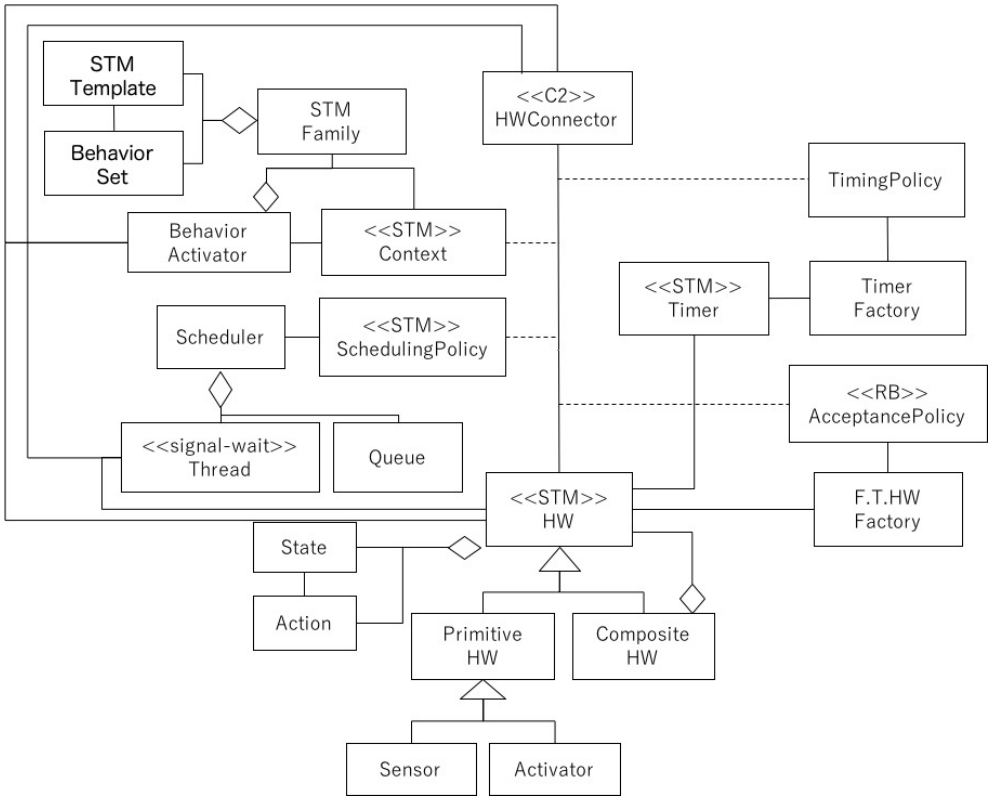
1. 理解容易なアーキテクチャ
2. 実現コードの標準化
3. 技術転換，すなわち，技術置換およびコード再利用が可能

以下，この利点を説明する．

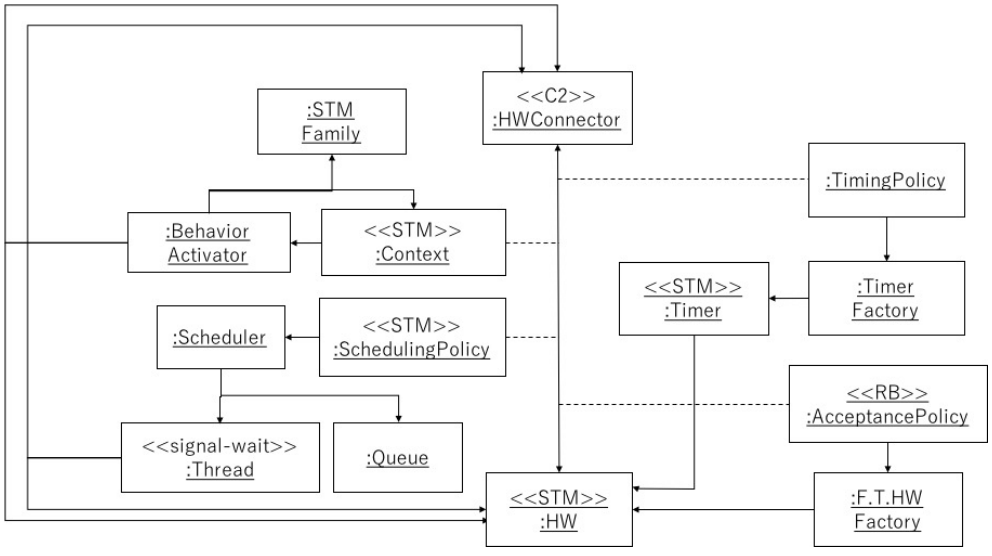
5.3.1 理解容易なアーキテクチャ

参照アーキテクチャ設計においては，PBR パターンを用いて非機能特性とコンテキストの横断的コンサーンを統一的に扱う構造が提示できた．動的振舞いに関しても PBR パターンだけを理解すれば，挙動を把握できる．

PBR パターンは，コアコンサーンと横断的コンサーンのモジュール分割のパターンであると同時に，分割されたモジュール群の協調に関連する記述を分離したものである．すなわち，コアコンサーンによって規定されるハードウェアの集合において，メッセージ通信の前後に横断的コンサーンによって規定されるモジュールへのメッセージ通信を付加する構造を示すものである．さらにすべての横断的コンサーンにおいても，協調に関連する記述や構造を標準化して定義した．このように定義したアーキテクチャにおいては，すべてのコンサーンについて，[21] で示されたアーキテクチャの利点を保証することとなる．本研究で設計した参照アーキテクチャは組込みソフトウェアに特徴的な非機能特性とコンテキストをハードウェア間の構造と分離して互いに独

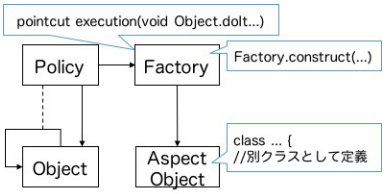


(a) 静的構造

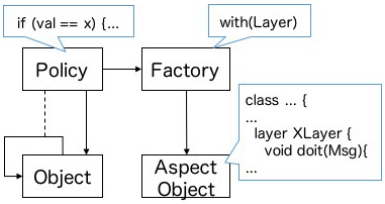


(b) 動的振舞い

図 5.15: 紙幣搬送システムの具象アーキテクチャ



(a) AspectJ による実現



(b) ContextJ による実現

図 5.16: 特定の言語を用いた実現

立に記述し，その織込み方法を示したものとまとめられる．

5.3.2 コードレベルで統一的な取り扱い

PBR パターンは，コードレベルでは，プログラミング言語のコード記述方式を定義するものである．特定の言語要素と PBR パターンのコンポーネントの対応関係からコードの標準化が可能となる．PBR パターンを適用しない場合は，アスペクト毎に適切なモジュール化がされず，その実現だけでなく変更が困難となることがある．例として，コンテキスト指向言語である ContextJ を用いた場合と，アスペクト指向言語である AspectJ を用いた場合の記述を PBR パターンとの関係を図 5.16(a)，(b) に示す．ContextJ では，PBR パターンの *Policy* は，レイヤを活性化させる条件式に相当し，*AspectObject* は，コンテキストに応じた振舞いを定義するレイヤに相当する．*Factory* による再構成は，ビヘイビアアクティベータに対応することから，これに対するメッセージ式としての with 文が相当する．AspectJ では，PBR パターンの *Policy* は，アドバイス記述に相当し，ここに状況に応じて *AspectObject* の生成を行なうための *Factory* へのメッセージおよび生成された *AspectObject* へのメッセージを記述することを規定している．この規定がなければ，アドバイス記述にオブジェクトに局所化されるべき処理をここにすべて記述することもでき，すべてのオブジェクトを代表するオブジェクトへのメッセージのみ記述することもできる．このように自由度が高いことから，コードが標準化されず，保守は難しくなる可能性がある．

5.3.3 既存の自己適応技術を説明可能

PBR パターンは，メタアーキテクチャパターンである．すなわち，そのコンポーネントであるファクトリやポリシー記述を入れ替えることで，[55] で参照されている既存のアーキテクチャに具体化できる．PBR パターンのファクトリの生成対象をコンポーネントとし，リアクティブに適応を実行するポリシーを記述すれば CASA[46] となる．一方，この生成対象をコンポーネント間の関連とすれば Rainbow[29] となり，プロアクティブに適応を実行するポリシーとして記述すれば K-Components[20] となる．

5.3.4 再利用の枠組み

PBR パターンを用いて，横断的コンサーンおよびコンテキストコンサーンについて独立して構造を導出することで，大きな粒度での再利用を可能とする利点を持つ．例えば，5.2.4 で示したように，Java の Thread

クラスライブラリを用いて並行性を実装する．この場合，各オブジェクトをメッセージキューでラッピングすることにより Java の Thread クラスライブラリがそのままコンポーネントとして再利用可能になる．

5.4 まとめ

組込みシステムは，並行に動作するハードウェアの集合によって実現される．それらの振舞いは，システムを取り巻く外部環境に応じて変化する．組込みシステムのプログラムの保守を容易にするためには，外部環境を反映したシステムの内部状態すなわちコンテキストをモジュールとして独立に管理できるようにすることが重要である．一方，組込みシステムの開発では，実時間性，耐故障性などの非機能特性を重視すべきであり，これらについてもモジュールとして独立に管理できるようにすることが重要である [1, 32, 40]．

PBR パターンを用いて組込みシステムのためのアスペクト指向アーキテクチャを設計し，コンテキスト指向計算とアスペクト指向計算を説明し，統一的にアスペクトとして実現した．動的振舞いに関しても PBR パターンだけを理解すれば，挙動を把握で きる．PBR パターンは，コードレベルでは，プログラミング言語のコード記述方式を定義するものである．PBR パターンに従うことでコードの標準化が可能となった．PBR パターンは，メタアーキテクチャパターンである．すなわち，そのコンポーネントであるファクトリやポリシー記述を入れ替えることで，既存のアーキテクチャに具体化できる．PBR パターンにより，アーキテクチャの理解が容易になるだけでなく，コンポーネントを再利用する枠組みが提案できた．

第 6 章

インタラクティブシステムのためのソフトウェアアーキテクチャ

インタラクティブシステムの開発支援のために，MVC やその派生のアーキテクチャスタイルが提案されてきた．これらのアーキテクチャスタイルは，オブジェクト指向によるモジュール分割に対していくつかの横断的コンサーンの分離を試みている．近年，インタラクティブシステムは，スマートデバイスのような移動体を対象とした場合，環境に応じて振舞いが変化する．本稿では，インタラクティブシステムのアーキテクチャ中心開発の基盤としてのアスペクト指向アーキテクチャを設計し，その有用性について議論する．自己適応のためのアーキテクチャパターンとして PBR パターンを定義し，このパターンを用いて横断的コンサーンを分離し，動的にアスペクトを付加する仕組みを設計した．アーキテクチャとアプリケーションの設計およびコードの理解，変更が容易になるだけでなく，ライブラリやミドルウェアを，大きな粒度で変更する枠組みが提供できた．

6.1 インタラクティブソフトウェアの開発の現状および問題点と解決策

インタラクティブソフトウェアは，Web アプリケーションとネイティブアプリケーションおよびこれらの組み合わせに分類される．Web アプリケーションは，Web ブラウザ内で稼働するプログラムとサーバ上で稼働するプログラムの協調により動作する．ネイティブアプリケーションは Web ブラウザを利用せずに使用者インタフェースを取り扱い，Web アプリケーションと同様にサーバ上のプログラムと協調して動作することが多い．

MVC モデルやその派生のアーキテクチャスタイルがインタラクティブシステムのために提案されてきた．これらのアーキテクチャスタイルでは，オブジェクト指向をコアコンサーンとし，横断的コンサーンの分離を試みている．インタラクティブシステムのアーキテクチャ中心開発では，アーキテクチャスタイルを適用して特定の実現技術に依存しない参照アーキテクチャを設計し，適用する実現技術を選択してそれを詳細化することで具象アーキテクチャを設計する．さらに具象アーキテクチャに基づいてアプリケーションの設計および実装を行なう．これら一連の開発において，適用するアーキテクチャスタイルや実現技術には様々な選択肢があり，それらの間には複雑な依存関係が存在するので，特定のアーキテクチャスタイルや実現技術を，異なる技術に転換することは一般に容易ではない．

本研究の目的は，インタラクティブシステムのアーキテクチャ中心開発において基盤となる共通アーキテクチャを設計することである．技術転換支援ならびに構造やコードの標準化を行なうべく，本研究では以下の 2 つの特徴的な方法をとった．

1. メタアーキテクチャとしての共通参照アーキテクチャを設計する
2. この共通参照アーキテクチャを設計に自己適応のためのメタパターンとして PBR パターンを用いる

ここでメタアーキテクチャとは，MVC やその派生など既存のアーキテクチャスタイルに基づく参照アーキテ

クチャを統一的に説明できるアーキテクチャを意味する．すなわち，それぞれのスタイルに基づく参照アーキテクチャはメタアーキテクチャとしての共通参照アーキテクチャから導出できるものである．共通参照アーキテクチャの設計にあたり，既存のアーキテクチャスタイルを調査し，横断的コンサーンを識別・分類した．これらを統合することで，特定のコンサーンを指定すれば既存のアーキテクチャスタイルを説明することができるようになる．共通参照アーキテクチャは，実現技術により特定できるコンサーンをパラメータとして，アーキテクチャを導出できる構造とした．これにより，実現技術の役割が明確となり，異なる実現技術との対応関係がアーキテクチャを介して理解可能となる．

レスポンシブウェブデザイン [43] を適用した場合や，スマートデバイスのような移動体を対象とした場合，環境に応じてアプリケーションの振舞いが変化する．この変化を PBR パターンを用いた動的再構成によって実現した．PBR パターンにより，アーキテクチャとアプリケーションの設計およびコードの理解，変更が容易になるだけでなく，ライブラリやミドルウェア等を，大きな粒度で変更する枠組みが提供できる．

6.2 アーキテクチャの設計

我々は，OASIS のアーキテクチャの定義 [42] が一般的なアーキテクチャの設計および運用の枠組みを定義しているとの認識に立ち，この定義に基づいてアーキテクチャについて議論する．この定義によると，参照モデルは，アーキテクチャ設計の基礎となるシステムの抽象表現である．具体的には，抽象化されたコンポーネントとそれらの関係を示すものである．参照アーキテクチャは参照モデルに従って定義される特定の技術に依存しないアーキテクチャであり，具象アーキテクチャは適用する実現技術を選択し参照アーキテクチャを詳細化したものである．アーキテクチャ設計にあたり，横断的コンサーンを統一的に扱うために，自己適応のためのアーキテクチャパターンとして PBR パターンを適用してアーキテクチャを設計する．

6.2.1 インタラクティブシステムのための参照モデル

SmallTalk[38] において MVC モデルが提案され，近年では，インタラクティブシステムのための参照モデルとして扱われるようになった．MVC モデルが十分に一般的であると考え，MVC モデルを参照モデルとして採用した．

6.2.2 参照アーキテクチャ

既存のアーキテクチャスタイルから複数の横断的コンサーンを識別し，参照アーキテクチャを設計する．

MVC の派生アーキテクチャ

MVC アーキテクチャとその派生として AM-MVC[57]，HMVC[9]，MVVM[59]，PAC[15]，MVP[53] について調査した [60]．

MVC アーキテクチャ (図 6.1) は，プレゼンテーションロジックからビジネスロジックを分離し，それぞれの独立な変更を可能にする．画面に表示される視覚的要素を扱う View，データとビジネスロジックを含むドメインモデルを扱う Model，使用者操作によるイベントを扱う Controller によって構成される．View と Model の関連は，変更を Model へのプッシュ通信または Model からのプル通信により認知し，画面を更新する ClassicMVC と，View と Model の依存関係を無くし，Controller からのプッシュ通信を画面更新のきっかけとすることを前提とした PassiveMVC がある．

AM-MVC(図 6.2) は，MVC の View もしくは Controller に定義されるプレゼンテーションロジックを分離し，View と Controller を，画面出力と使用者入力処理と画面構築の役割に分ける．MVC から分離したプレゼンテーションロジックの実行を役割としてもつ ApplicationModel を追加している．

HMVC(図 6.3) と PAC(図 6.4) は，階層構造を用いて記述することを目的としている．HMVC は，MVC

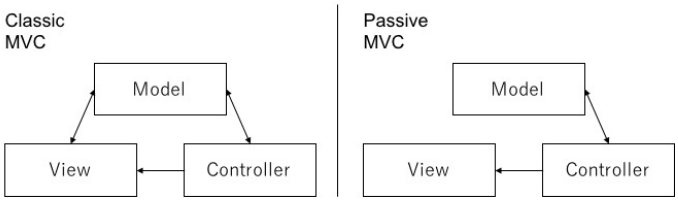


図 6.1: Model-View-Controller(MVC)

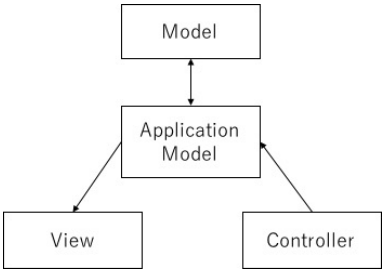


図 6.2: ApplicationModel-Model-View-Controller(AM-MVC)

の Controller 同士で階層間の協調を実現する．PAC は，その構成単位を特定の機能を実現する Agent としている．Agent は，画面出力と使用者入力処理を扱う Presentation，Agent の機能とデータを扱う Model，これらの間の協調と階層間の協調を実現する Control によって構成される．

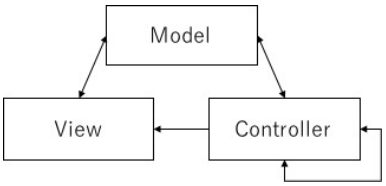


図 6.3: Hierarchical-Model-View-Controller(HMVC)

MVP(図 6.5) は，リアクティブシステムに特化した派生である．画面に表示される視覚的要素と使用者操作によるイベントを扱う View，プレゼンテーションロジックを扱う Presenter，ドメインモデルを扱う Model によって構成される．MVP の View は使用者操作によるイベント処理と画面出力を行なうことから，MVC の View と Controller に対応する．MVP の Presenter は，MVC の View と Controller に横断する．Presenter と Model，View と Model 間はすべてイベント通知で協調する．これら要素間の関連も，MVC と同様に Classic タイプと Passive タイプに分類される．

MVVM(図 6.6) は，プレゼンテーションロジックとビジネスロジックの分離を目的とした，MVC とは異なる分割によるアーキテクチャである．画面に表示される視覚的要素と使用者操作によるイベントを扱う View，ドメインモデルを扱う Model，View と Model を関連づけ外部表現とドメインモデルを結合した View-Model によって構成される．Abstraction は，特定の機能に関連する MVC の Model に対応する．

インタラクティブシステムの横断的コンサーン

6.2.2 で述べたの既存のアーキテクチャスタイルが分離を試みている横断的コンサーンは以下の通りである．

1) 制御コンサーン
2) 表示コンサーン
3) UI コンサーン
4) 表示モデルコンサーン

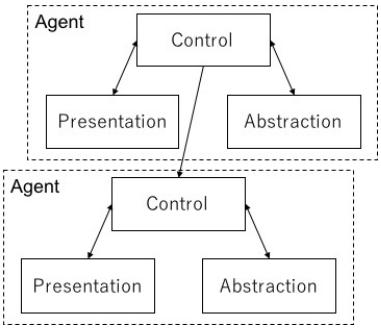


図 6.4: Presentation-Abstraction-Control(PAC)

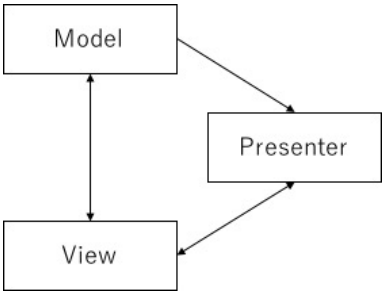


図 6.5: Model-View-Presenter(MVP)

5) 階層化コンサーン

これら横断的コンサーンとアーキテクチャスタイルの関係を表 6.1 示す．MVC，AM-MVC，HMVC では，入力に関するコンサーン (制御コンサーン) に基づき分割を行なっている．MVC，AM-MVC，HMVC では，画面表示 (表示コンサーン) に基づき分割を行なっている．MVP，PAC，MVVM では，入出力に関するコンサーン (UI コンサーン) に基づき分割を行なっている．AM-MVC，MVVM では，画面表示用に加工されたデータモデルに関するコンサーン (表示モデルコンサーン) に基づき分割を行なっている．HMVC，PAC では，階層関係を規定するコンサーン (階層化コンサーン) に基づき分割を行なっている．

これらの調査をもとに，アーキテクチャスタイルとこれらの横断的コンサーンの組合わせを精査した結果，横断的コンサーンを直行する 2 つの次元で分類できた．表 6.2 に，各次元の組み合わせと導出されるアーキテクチャスタイルを示す．縦軸と横軸はそれぞれの次元における横断的コンサーンを示し，これの組み合わせによって導出されるアーキテクチャスタイルを示している．例えば， AM-MVC は，オブジェクト指向に対して制御と表示，表示モデルコンサーンを分離している．

表 6.1: 横断的コンサーンとアーキテクチャスタイルとの関係

コンサーン	アーキテクチャスタイル
制御	MVC，AM-MVC，HMVC
表示	MVC，AM-MVC，HMVC
UI	MVVM，MVP
表示モデル	AM-MVC， MVVM
階層化	HMVC，PAC

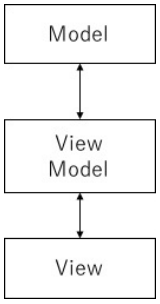


図 6.6: Model-View-ViewModel(MVVM)

表 6.2: 各次元の選択で導出されるアーキテクチャスタイル

次元	表示モデル	階層化	なし
制御, 表示	AM-MVC	HMVC	MVC
UI	MVVM	PAC	MVP

前述の横断的コンサーンの分離を目的 (インテント) として，(ベース) パターンを導出する．PBR パターンのコンポーネントとベースパターン導出のために与える役割の関係を表 6.3 に示す．このベースパターンを適用し，コンポーネントの振舞いを具体化することでアーキテクチャを設計する．

表 6.3: PBR パターンのコンポーネントに与える役割

コンサーン	PBR パターンのコンポーネント	役割
制御	Policy	EventListener
	Factory	HandlerFactory
	Aspect Object	EventHandler
表示	Policy	ViewTransitionPolicy
	Factory	DisplayImageConstructorFactory
	Aspect Object	DisplayImageConstructor
UI	Policy	ViewTransitionPolicy
	Factory	UIFactory
	Aspect Object	UIComponent
表示モデル	Policy	ViewModelPolicy
	Factory	ViewModelFactory
	Aspect Object	ViewModel

メタアーキテクチャの概要

前節で整理した横断的コンサーンによって規定されるアスペクトをすべて統合し，メタアーキテクチャとした．その概略を図 6.7 に示す．横断的コンサーンを選択し，図中の対応するアスペクトと関連を残すことで，アーキテクチャを導出する．使用者インタフェース (*UI*) は，制御 (*Controller*) と表示 (*View*) の複合コンサーンであることから，これらを内包するものとして表現している．

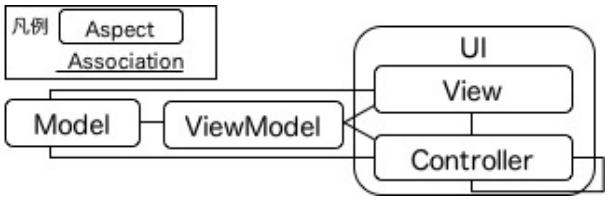


図 6.7: メタアーキテクチャの概略

以下より，コアコンサーンと，横断的コンサーンによって規定されるアスペクトの構造について説明する．

モデル (コアコンサーン)

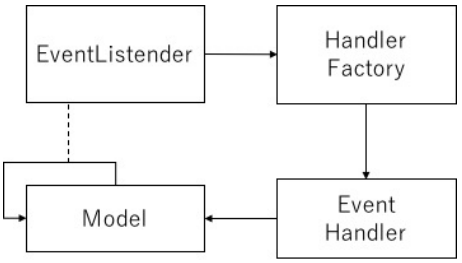
オブジェクト指向に基づいて設計されるモデルは，アプリケーション依存なので，その構造の共通性を抜き出してアーキテクチャで規定することはできない．参照アーキテクチャとしては，コンサーンの存在を定義しているだけで，その構造は提示しない．アプリケーションに依存する構造を抽象化すると，一般にオブジェクトとこれらの間のメッセージ通信となる．アプリケーションが扱うデータをオブジェクトの集合としてモデル化し，横断的コンサーンをアスペクトとして分離する．

制御 (Controller) コンサーン

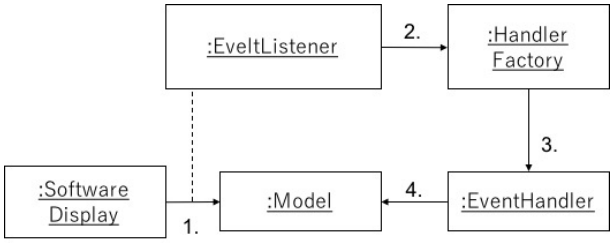
制御コンサーンは入力処理とモデルにおけるビジネスロジックとの分割を規定する．移動体を考えた場合，その位置情報に応じて入力イベントの取り扱いを動的に変更する必要があることから，検知されるイベントとこの取り扱いの関係を動的に再構成させる必要がある．この静的構造と動的振舞いを図 6.8(a)，(b) に示す．PBR パターンを適用し，イベントリスナ (*EventListener*)，イベントハンドラ (*EventHandler*)，ハンドラのインスタンスを生成するファクトリ (*HandlerFactory*) から構成した．*Object* に対するインスタンス生成のメッセージを横取りし，これら構成要素のインスタンスを生成する．*EventListener* は，使用者からの入力イベント (以下，外部イベント) を検知し，アプリケーション内部でのイベント表現 (以下，内部イベント) に変換する．*HandlerFactory* は *EventHandler* のインスタンスを生成する．*EventHandler* は内部イベントを適切なオブジェクトに通知する．

表示 (View) コンサーン

表示コンサーンは表示処理とモデルにおけるビジネスロジックとの分割を規定する．様々な画面サイズのデバイスを考えた場合，そのサイズに応じて画面遷移や出力画面を動的に再構成させる必要がある．この静的構造と動的振舞いを図 6.9(a)，(b) に示す．PBR パターンを適用し，画面遷移を管理するポリシー (*ViewTransitionPolicy*)，特定の表示画面の具象表現を構築するコンストラクタ (*DisplayImageConstructor*)，このコンストラクタを生成するファクトリ (*DisplayImageConstructorFactory*) から構成した．*Object* 間のメッセージを横取りし，*ViewTransitionPolicy* で，*DisplayImageConstructorFactory* に特定の具象表現を構築するための *DisplayImageConstructor* を生成させる．*DisplayImageConstructor* は，特定の具象表現 (*DisplayImage*) を構築する．

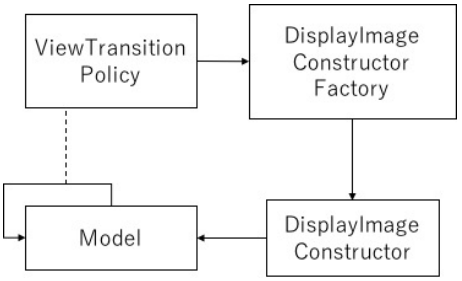


(a) 静的構造

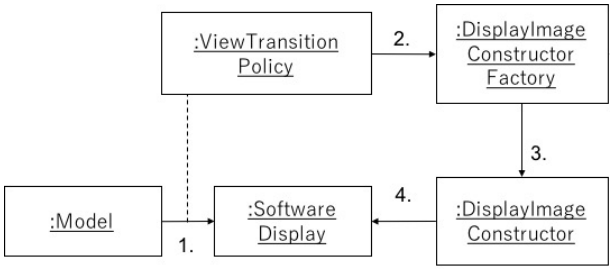


(b) 動的振舞い

図 6.8: 制御 (Controller)



(a) 静的構造

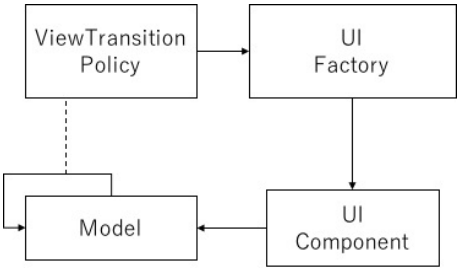


(b) 動的振舞い

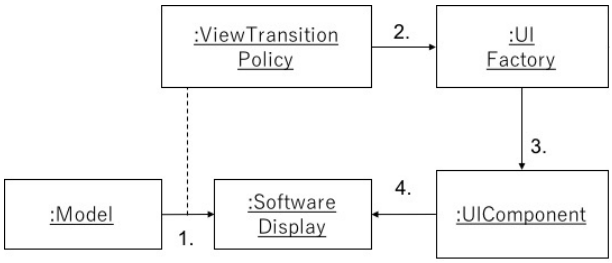
図 6.9: 表示 (View)

UI(使用者インタフェース) コンサーン

UI コンサーンは入出力処理とモデルにおけるビジネスロジックとの分割を規定する．制御，表示コンサーンのように，移動体として考えた場合や，様々な画面サイズのデバイスを考えた場合，画面遷移や出力画面の関係を動的に再構成させる必要がある．この静的構造と動的振舞いを図 6.10(a)，(b) に示す．PBR パターンを適用し，画面遷移を管理するポリシー (*ViewTransitionPolicy*)，入出力の責務を持つ UI コンポーネント (*UIComponent*)，UI コンポーネントを構築するコンストラクタ (*UIFactory*) から構成した．*Object* 間のメッセージ通信を横取りし，*ViewTransitionPolicy* で，*UIConstructor* に表示画面の *UIComponent* を生成させる．使用者はこの *UIComponent* を介して，イベントを *Object* に通知する．このアスペクトは，上で述べた制御アスペクトと表示アスペクトを合成したアスペクトである．



(a) 静的構造



(b) 動的振舞い

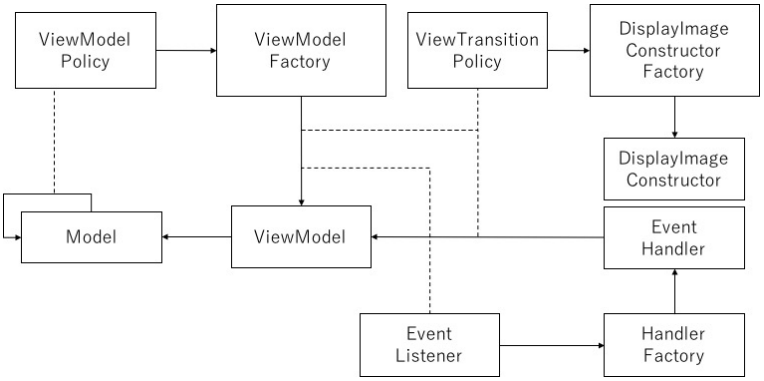
図 6.10: UI(使用者インタフェース)

表示モデルコンサーン

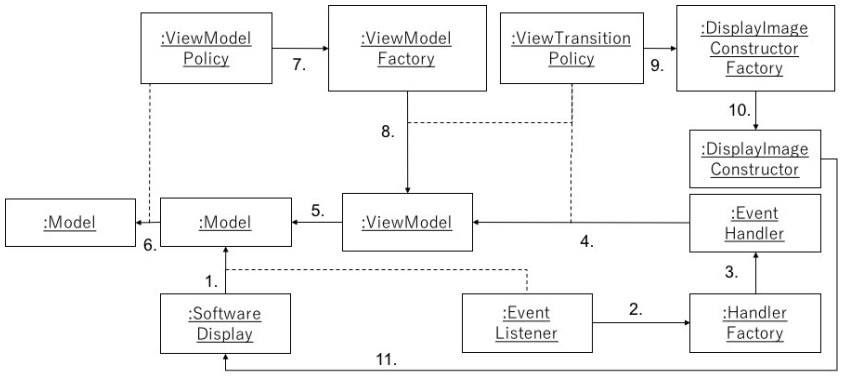
表示モデルコンサーンはモデル内のすべてのオブジェクトに横断し，それらと画面表示用のデータモデルとの分割を規定する．表示画面の抽象表現を定義し，具象表現を独立して切替え可能にすることを目的として設計する．表 6.2 に示したように，表示モデルコンサーンは，異なる次元で選択した横断的コンサーンに応じてその構造が異なる．UI コンサーンを選択したさいの静的構造と動的振舞いを図 6.11(a)，(b) に示す．制御コンサーンと表示コンサーンを選択したさいの静的構造と動的振舞いを図 6.11(a)，(b) に示す．表示モデルアスペクトを，構築する表示モデルを決定するポリシー (*ViewModelPolicy*)，ビューの抽象表現としての表示モデル (*View Model*)，表示モデルを生成するファクトリ (*ViewModelFactory*) から構成した．*Object* 間のメッセージ通信を横取りし，*ViewModelPolicy* で，*ViewModelFactory* に *ViewModel* を生成させる．

6.2.3 具象アーキテクチャ

具象アーキテクチャは，実現技術を選択し，参照アーキテクチャの構造を詳細化したものである．実現技術とは，製品固有のモジュール構成法，プロトコル，および，適用するコード記述方法を指す．インタラクティブシステムは Web アプリケーションとネイティブアプリケーションに大別される．ここでは，現在主流であ

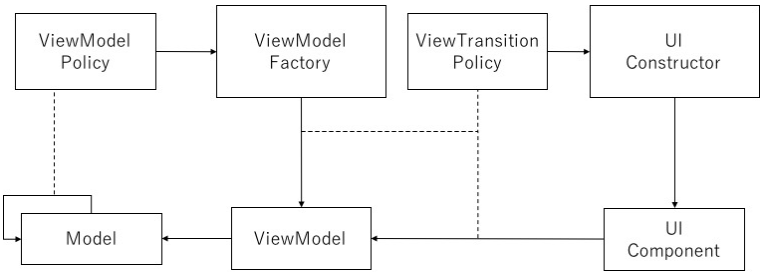


(a) 静的構造

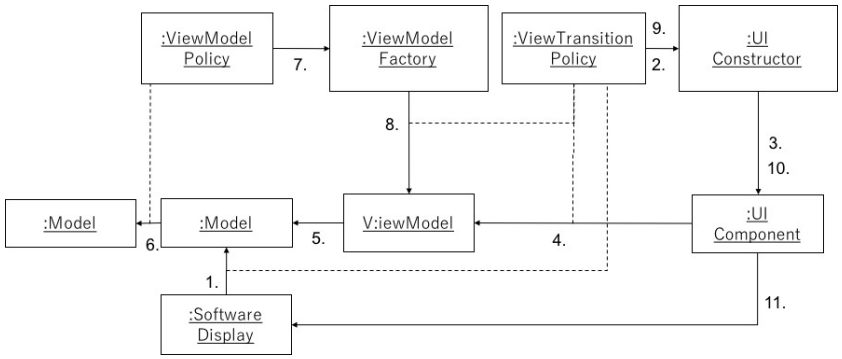


(b) 動的振舞い

図 6.11: 表示モデル (制御，表示コンサーン選択)



(a) 静的構造



(b) 動的振舞い

図 6.12: 表示モデル (UI コンサーン選択)

る Web アプリケーションを設計するための具象アーキテクチャを説明する．

ここでは，Web アプリケーションのアプリケーションフレームワークとして Struts2 を用いて実現することを念頭に置いている．Struts2 は制御，表示，表示モデルコンサーンを分離した構造を前提としている．これらの横断的コンサーンをパラメータとして，メタアーキテクチャに与えて導出される参照アーキテクチャ (図 6.11(a)，(b)) を前提とした具象アーキテクチャについて説明する．

インタラクティブシステムに適用される実現技術

本研究で対象とするインタラクティブシステムの実現技術を整理すると以下の通りである．プロトコルおよびコード記述方法は，アプリケーションの種類によって異なる．

- **モジュール構成法**
  インタラクティブシステムはリアクティブシステムなので，イベントとその処理を対にして記述しなければならない．それらを平たく記述すると case 文等となるが，ネストが深くなると保守が困難となる．したがって，このようなモジュール構成は適切ではなく，これらは状態遷移機械としてモデル化することが適切である．
- **プロトコル**
  インターネットを介した通信プロトコルとして，Web アプリケーションでは HTTP がある．ネイティブアプリケーションではアプリケーション内でプロトコルを決定して実現する．
- **コード記述方法**
  具象表現形式は，Web アプリケーションでは標準的なものに HTML5 および CSS3 がある．ネイティブアプリケーションではアプリケーションフレームワークが独自のものを提供している．アプリケーションフレームワークは，Web アプリケーションでは多数提案されているが，例えば，Struts2 や Ruby on Rails が挙げられる．

ここでは一般的な場合の一例として実現技術を選択する．具象表現形式として，HTML5 および CSS3 を選択した．通信プロトコルには，HTTP を選択した．モジュール構成法として状態遷移モデルを選択した．

実現技術の組み合わせは従属的である．すなわち，数多くあるプログラミング言語や実現技術を交換的に用いることは不可能である．例えば，プログラミング言語として Java を選択した場合，Struts2 を用いることができるが，C#を選択した場合は，ASP.NET を用いることができる．事例の Web アプリケーションでは，Java を用いて実現することを念頭に置いているので，アプリケーションフレームワークとして Struts2 を用いる．

以上をまとめると，次の実現技術を選択した．

- **プログラミング言語：**Java
- **モジュール構成法：**状態遷移機械としてモデル化
- **具象表現形式：**HTML5，CSS3
- **外部との通信のプロトコル：**HTTP
- **外部イベント表現形式：**URL クエリパラメータ
- **内部イベント表現形式：**イベントクラス (Java 標準ライブラリ)
- **アプリケーションフレームワーク：**Struts2

状態遷移機械によるモデル化

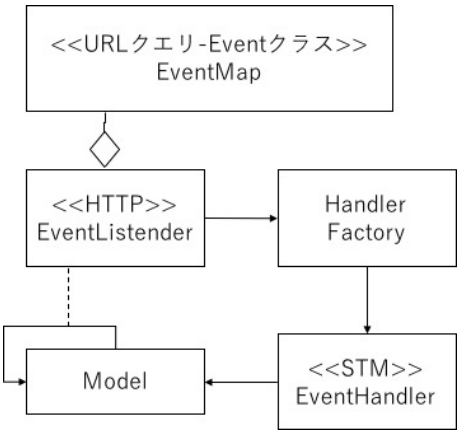
ソフトウェアの保守性を考慮し，ミーリ型状態遷移機械を導入する．すなわち，現在の状態に応じたイベントとアクションの対を定義する．ミーリ型状態遷移機械に対する変更として，状態およびアクションや状態遷移の変更を独立して行なうことを目的として，状態とアクションを別のモジュールとして定義した (*State*,

Action)．これによりモジュールの独立性を確保する．状態 (*State*) はイベントに応じてアクション (*Action*) を実行し，遷移後の状態 (*State*) を返すことで状態遷移を表現する．

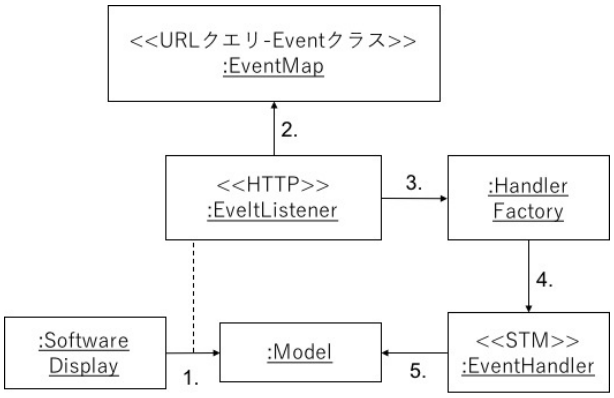
これらは，オブジェクト指向コンサーン，制御コンサーン，表示コンサーンに対して導入し，適用されるコンポーネントにステレオタイプで <<STM>> と示す．

制御 (Controller) **コンサーン**

外部イベントと内部イベントとの変換を実現するために, このイベント間での対応関係を管理する *EventMap* を導入した．状態に応じて受理可能なイベントを管理するために，*EventHandler* を状態遷移機械としてモデル化する．状態遷移機械に対して前述と同様に変更を独立して行なうことを目的として，静的構造と動的振舞いを設計する．静的構造と動的振舞いを図 6.13(a)，(b) に示す．*MappingRule* を変更するだけで，実現技術として選択した外部イベントを切替えることができる．



(a) 静的構造

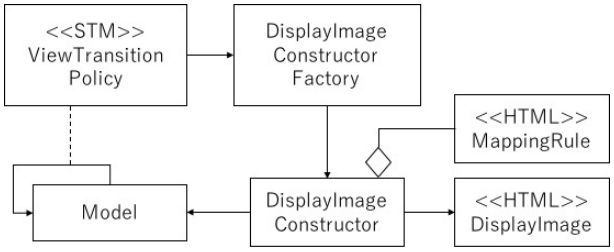


(b) 動的振舞い

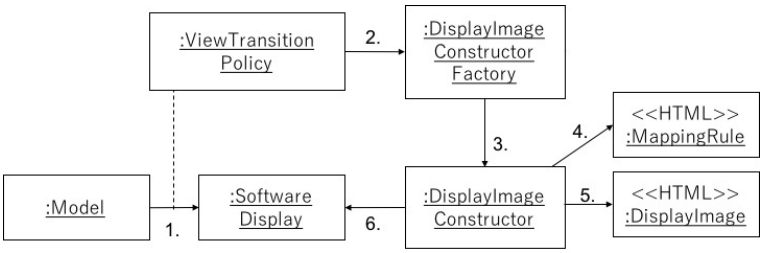
図 6.13: 制御 (Controller)

表示 (View) **コンサーン**

画面内部表現と具象表現との変換を実現するために，この表現間での変換規則を定義した *MappingRule* を導入した．現在表示中の画面を状態として捉え，*ViewTransitionPolicy* を状態遷移機械としてモデル化する．状態遷移機械に対して前述と同様に変更を独立して行なうことを目的として，静的構造と動的振舞いを設計する．静的構造と動的振舞いを図 6.14(a)，(b) に示す．*MappingRule* を変更するだけで，実現技術として選択した出力画面の具象表現形式を切替えることができる．



(a) 静的構造



(b) 動的振舞い

図 6.14: 表示 (View)

表示モデルコンサーン

具象表現形式の標準である HTML，CSS を参考にして，これらを抽象化することで *ViewModel* の詳細構造を定義した．内容と役割と見栄えに関する *ViewContent*，*DisplayImageContent*，*Style* を定義した．この静的構造を図 6.16 に示す．その他の構造については，図 6.16 と同じなのでここでは省略する．

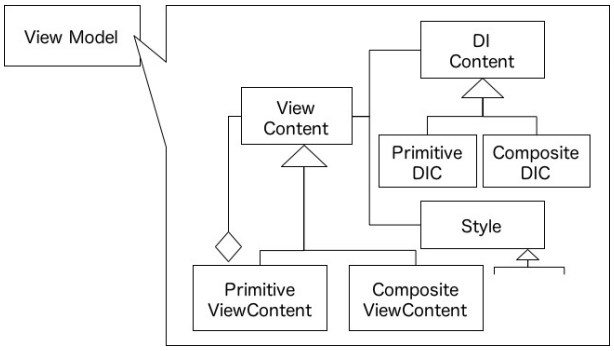
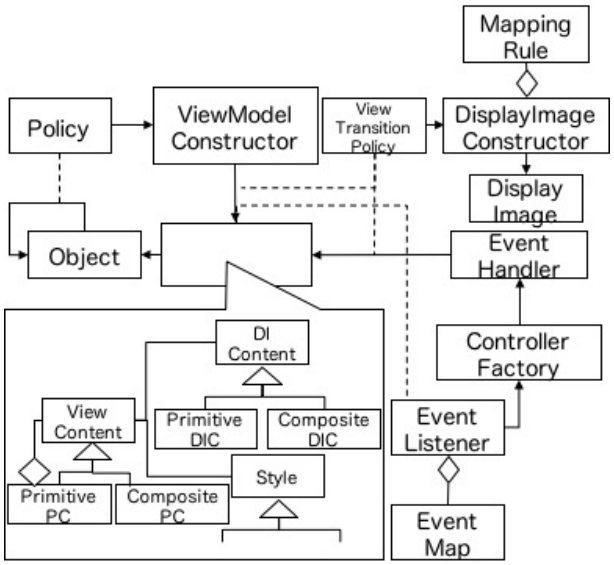


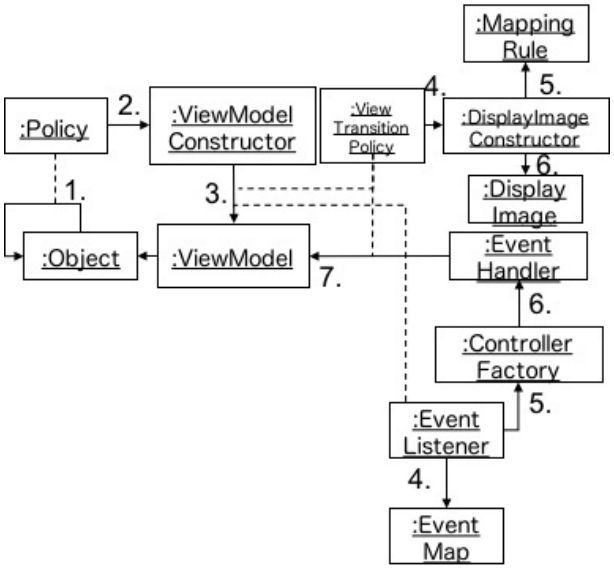
図 6.15: 表示モデルの詳細構造

具象アーキテクチャのまとめ

ここでは，Web アプリケーションを例として具象アーキテクチャを示した．提案するアーキテクチャは，異なるアプリケーションについても説明可能なものとして設計できた．例えば，Java によるネイティブアプリケーションの実現を想定した場合，外部表現形式は標準 GUI ライブラリとして用意されている Swing，イベントの表現形式はイベントクラスを選択すれば良い．この時の具象アーキテクチャは，表示コンサーンの MappingRule，制御コンサーンの EventListener がこれら実現技術を適用したコンポーネントとなる．



(a) 静的構造



(b) 動的振舞い

図 6.16: 表示モデルコンサーン

6.3 考察

本研究では，既存のアーキテクチャスタイルを分類，整理してインタラクティブシステムのためのアスペクト指向アーキテクチャを設計した．このアーキテクチャおよび PBR パターンによる利点は以下の 4 通りである．

1. メタアーキテクチャにより既存のアーキテクチャスタイルを説明可能
2. コードレベルでの統一的な取り扱いが可能
3. 大きな粒度での再利用が可能
4. ソフトコンピューティングの実現の可能性

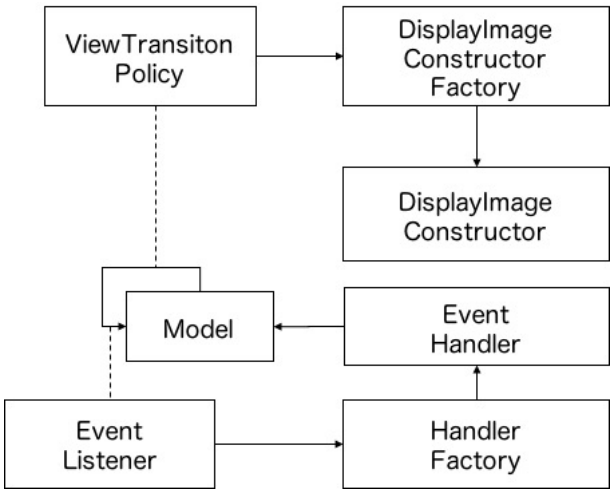


図 6.17: 制御コンサーンと表示コンサーンを指定して導出された MVC アーキテクチャ

6.3.1 メタアーキテクチャとしての参照アーキテクチャ

提案したメタアーキテクチャは，横断的コンサーンを指定し，アスペクトとして分離することにより特定のアーキテクチャスタイルに基づく参照アーキテクチャを導出する．6.2.3 で示した例とは異なる例として，表示コンサーン，制御コンサーンのみを選択した場合，図 6.17 のように，MVC アーキテクチャが構築できる．6.2.2 で示したアーキテクチャスタイルについて説明可能であることを表 6.2 にまとめている．

メタアーキテクチャを理解すれば，特定のアーキテクチャスタイルを理解できるだけでなく，特定の技術に習熟した開発者は，このアーキテクチャを介して異なる技術について類推し，技術転換を図ることが可能となる．特定の具象アーキテクチャは，それぞれのコンポーネントと実現技術との関係が明らかとなっている．具象アーキテクチャは，前提とする参照アーキテクチャとの関係が明らかとなっている．メタアーキテクチャでは，異なる参照アーキテクチャ間でのコンポーネント間の関係が明らかとなっている．したがって，メタアーキテクチャを介して特定のアーキテクチャの実現に用いられる技術から，異なるアーキテクチャの実現に用いる技術を理解することができる．

6.3.2 コードレベルでの統一的な取り扱い

PBR パターンは，コードレベルでは，プログラミング言語のコード記述方式を定義するものである．PBR パターンに従うことでコードの標準化が可能となる．このパターンを適用しない場合は，アスペクト毎に適切なモジュール化がされず，新規のコード記述だけでなく変更も困難となることがある．PBR パターンの *Policy* は，アドバイス記述に相当し，ここにコンテキストに応じて *AspectObject* の生成を行なうためのメッセージおよび生成された *AspectObject* へのメッセージを記述することを規定している．この規定がなければ，オブジェクトに局所化されるべき処理をアドバイス記述にすべて記述することもでき，すべてのオブジェクトを代表するオブジェクト (デザインパターンにおけるファサードパターン [28]) へのメッセージのみ記述することもできる．このように自由度が高いことから，コードが標準化されず，その保守は難しくなる可能性もある．例えば，表示コンサーンの実現では，図 6.18 に示すように，画面遷移に関連する記述と具象表現に変換する記述がアドバイス記述として実現される．この時，画面遷移の独立した変更や，他の具象表現への変換に変更することが困難である．

```
after() : build_ViewModel() returning(Msg msg) {
  if (this.currentPage.equals("Start")) { //ページ遷移管理
    Node root = msg.getViewModel();
    String html = "";
    for (Node n : root.getChildren()) { //内部表現走査
      if (n == Page ) {...                    //外部表現変換
      } else if (n == Header) { ...
      } else if (n == Title) { ...
      }
    }
    display.output(html);
  } else if (this.currentPage.equals("Menu")) { ...
}
```

図 6.18: PBR パターンを適用しない場合の実現

6.3.3 大きな粒度での再利用

6.2.3 で設計した具象アーキテクチャにおける，コンポーネント *EventListener*，*DisplayImageConstructor* について再利用が可能である．

EventListener については，図 6.19(a) に示すように，特定の実現技術を用いた実装は多相型で定義される．6.2.3 で示した例では，HTTP 通信の実現として，ServerSocket クラスや HttpServlet クラスを用いて実現し，上の例では，Swing の ActionListener クラスを用いて実現する．これらは，PBR パターンを適用して定義された *EventListener* に内包され，これを再利用することができる．

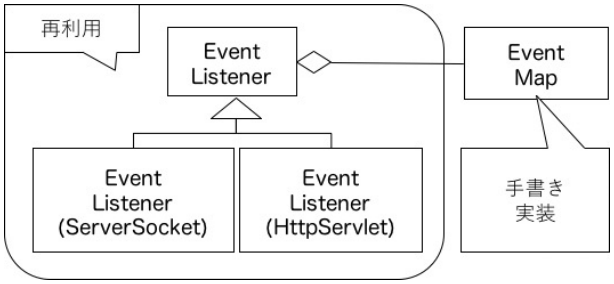
DisplayImageConstructor については，*MappingRule* がラッピングコードに相当し，これの小規模な書換えにより再利用が可能となる．例えば，Ruby on Rails における ERB テンプレートや，Java の JSP ではこの処理系が *DisplayImageConstructor* に対応し，*MappingRule* に抽象表現とテンプレートとの対応関係を記述することにより *DisplayImageConstructor* を再利用することができる．

6.3.4 ソフトコンピューティングの実現の可能性

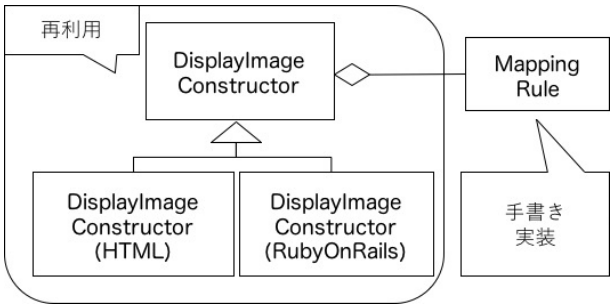
PBR パターンを用いることで，ハードコンピューティングやソフトコンピューティングなどの計算方法の実現を PBR パターンのコンポーネントの具象化の問題として扱うことが可能となる．コンポーネントに役割を与えることにより，特定の計算方法のための構造が得られる．以下，PBR パターンを用いたハードコンピューティングおよびソフトコンピューティングの実現について述べる．

CSA/I-Sys は，PBR パターンのポリシーコンポーネントを，ハードコンピューティングによる再構成のためのポリシーとして具象化可能とした．例えば，レスポンシブ Web デザイン [43] のような表示画面の再構成に対応可能である．利用しているデバイスの画面サイズをコンテキストとし，このコンテキストに応じて *ViewConstructor* のインスタンスを変更する．*ViewConstructor* は，特定のデバイスの画面サイズ用の *ViewModel* を構築する．結果として，画面サイズに応じた *DisplayImage* が生成される．

PBR パターンのポリシーは，学習アルゴリズムによる再構成を行なうソフトコンピューティングポリシー



(a) EventListener の再利用



(b) DisplayImageConstructor の再利用

図 6.19: 再利用の例

として実現可能であると考えている．すなわち，学習によって最適な画面表示を作り出すことが可能になる．例えば，ユーザの操作履歴をコンテキストとし，このコンテキストを基にポリシーはユーザの嗜好を学習する．この学習結果から最適な *ViewConstructor* の選択や，*ViewConstructor* の生成を行なう．

今後，新たに提供されるデバイスや，全てのユーザのニーズを事前に想定することは困難である．我々は，再構成をソフトコンピューティングによって実現することで，より使用性の高いアプリケーションの実現が可能となると考える．PBR パターンは，ハードコンピューティングからソフトコンピューティングまで多種のインタラクティブソフトウェアのソフトウェアプロセスを統一的に説明する枠組みを提供するものとして用いることができる．

6.4 まとめ

インタラクティブソフトウェアの実行時環境と開発環境は多岐にわたることから，環境の差異が生産性向上の障壁となっている．インタラクティブソフトウェアの共通アーキテクチャを提案することでこの問題の解決を試みた．共通参照アーキテクチャの設計においては，MVC アーキテクチャとその派生である既存のアーキテクチャを調査し，それらのアーキテクチャが分離を試みている横断的コンサーンを特定することで，アスペクト指向アーキテクチャとして統合した．具象アーキテクチャは，実現技術を選択し，共通参照アーキテクチャを詳細化することにより設計した．

アスペクト指向アーキテクチャとして共通アーキテクチャを設計することで，幾つかの既存の参照アーキテクチャと既存の具象アーキテクチャを説明可能にすることができた．コアコンサーンによって規定される分割に対して矛盾するコンサーンについて，アスペクトとして分離することで，これらを矛盾なく説明可能となった．共通アーキテクチャをプロダクトラインアーキテクチャとしたインタラクティブシステムのプロダクトライン開発を実現することで，コア資産間の追跡性が確保されると考える．また，MDA に基づくことで，PIM を既存のアーキテクチャ，PSM をフレームワークとし，入力される仕様に応じてアプリケーションフレームワークの自動生成が可能となる．現状では，全ての事例について共通アーキテクチャが説明可能であることを

確認していない．

特定の技術に習熟した開発者は，このアーキテクチャを介して異なる技術について類推し，技術転換を図ることが可能となる．コードレベルにおいても，この PBR パターンを理解するだけで PBR パターンの取り扱いの問題として，その設計や実装を議論することが可能である．PBR パターンに定義されるコンポーネントに対して実現技術を対応つけて，*EventListener*，*MappingRule* について再利用できることを確認した．このように大きな粒度で再利用可能となったのは，PBR パターンによってアスペクトの構造が標準化された結果，定義される責務から再利用コンポーネントを定義する手がかりを得やすくなったことによる．

第 7 章

ソフトウェアアーキテクチャに基づくプログラムの自動生成

本稿では，モデルコンパイラのための一般的なソフトウェアアーキテクチャについて説明する．このアーキテクチャは，モデルコンパイラおよびメタモデルコンパイラにおける統一的な構造を示す．我々は，グラフからグラフへのモデル変換のための一般パターンをアーキテクチャの基礎とし，テキストからグラフ，グラフからテキストへの変換を考慮してこのパターンを拡張する．アスペクト指向の概念を導入することにより，グラフからテキストへの変換および/またはグラフからテキストへの変換を，グラフからグラフへの変換に柔軟に織り込むアーキテクチャを設計することを可能とした．

我々は，このアーキテクチャはメタレベルのツールにも適用可能であることも示す．すなわち，このアーキテクチャに基づき，メタモデルコンパイラを開発可能である．モデルコンパイラのプログラムを生成するために，ソースモデルからターゲットモデルへのモデル変換規則の構文規則を入力とする．

インタラクティブソフトウェアのモデルコンパイラを作成する事例を通じて，アーキテクチャの有用性と実用性を考察する．

7.1 背景

MDE（Model-Driven Engineering）[26, 56] の概念は，実際のソフトウェア開発で広く適用されており，MDE に基づくツールはさまざまなベンダーによって提供されている．モデル変換は MDE で行われる最も重要なアクティビティであり，様々なモデルコンパイラを統合し，実現のための系統的な方法を整理することは重要である．

ソフトウェア開発に適用可能なモデルコンパイラを作成するには，既存のグラフベースのモデルの変換手続きを再利用するだけでなく，いくつかの解決が必要になることがよくある．モデルのテキスト表記形式として，プログラミング言語，仕様言語，および DSL（ドメイン固有言語）など数多くの形式がある．このような形式で記述されたテキストをモデルコンパイラの入力する場合，グラフベースのモデル変換が行われる前に入力を解析する機能が必要となる．また，モデルコンパイラは，変換されたグラフベースのモデルを特定の形式で記述されたテキストに変換し，出力することが求められる場合がある

開発者は，モデルコンパイラを開発するさいに次の 3 つの変換手続きについて考慮する必要がある．

1. テキストをグラフ表記のソースモデルへ変換するためのテキスト - グラフ手続き
2. グラフ表記のソースモデルをグラフ表記のターゲットモデルに変換するグラフ - グラフ手続き
3. グラフ表記のターゲットモデルをテキストに変換するグラフ - テキスト手続き

本研究では，モデル変換ツールとしてのモデルコンパイラに共通のソフトウェアアーキテクチャを設計する．このアーキテクチャは，いくつかのモデルコンパイラを抽象化し，一般化することで定義した．本研究は，グラフ - モデル変換の一般的なパターン [39, 33] をアーキテクチャの基礎とし，テキスト - グラフ変換とグラフ

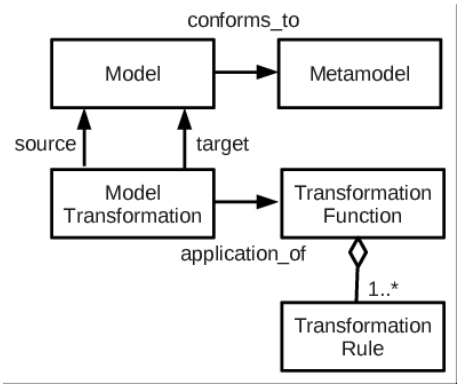


図 7.1: モデルコンパイラの構造の概要

- テキスト変換を考慮して拡張する．これらテキストからグラフへの変換やグラフからテキストへの変換を，グラフからグラフへの変換に柔軟に織込み可能なアスペクト指向のアーキテクチャを設計する．

提案するアーキテクチャはメタレベルのモデルコンパイラにも適用可能であることを示す．すなわち，モデルコンパイラのプログラムを作成するために，モデル変換規則およびソースモデルとターゲットモデルの両方の構文ルールを入力としたメタモデルコンパイラをアーキテクチャに基づくことで実現可能である．

一般にソフトウェアアーキテクチャはその構造を定義するだけでなく，その構造は一連の系統的な生産プロセスも含意する [4]．本研究の提案するアーキテクチャは，パーサジェネレータやグラフ変換などの既存の技術の柔軟な選択と統合による系統的な開発を可能とする．

組込みソフトウェアのためのモデルコンパイラやこの開発に関する事例を通じて本研究のアーキテクチャの有用性と実用性について考察する．

結果として，様々なモデルコンパイラのための共通のソフトウェアアーキテクチャを設計し，このアーキテクチャに基づいたモデルコンパイラを作成するための関連技術を統合するための系統的プロセスを提案したことである．このアーキテクチャはメタレベルのアーキテクチャーに適用可能であり，モデルコンパイラの生成が実現可能となる．

7.2 モデルコンパイラの開発における課題

MDE では，一連の変換をモデルに適用してソフトウェア開発を行なう．さまざまな分野のソフトウェアの開発において，MDE に基づくモデル変換アプローチ [17] が適用されている．図 7.1 は，モデル，モデル変換，および関連する概念 [45] の抽象化した構造を示す．モデル変換系 (*Model Transformation*) の実行は入力としてソースモデルを取り，それを処理してターゲットモデルを生成する．ソースまたはターゲットのいずれかの各モデルの構文は，メタモデル (*Metamodel*) によって規定される．変換プロセスは，変換規則 (*Transformation Rules*) のセットによって定義される変換手続き (*Transformation Function*) を適用することによって実行される．

MDE でのモデル変換の概念は，UML[8] で記述されたものなど，グラフベースのモデルを処理することに主に目的としている．MOF[52] で書かれたメタモデルは，モデルの抽象構文を定義する．すなわち，グラフからグラフへのモデル変換の場合，図 7.1 のモデル (*Model*) とメタモデル (*Metamodel*) との間の関連 (conforms_to) は，各モデル要素のタイプのみがメタモデル内で定義されていることを示すインスタンス化関係を表す．グラフからグラフへのモデル変換は，さまざまな方法で実装される．Java，C などの汎用プログラミング言語，ATL[33]，QVT[51] などの特定の言語を用いて変換規則を指定することができる．また，仕様にグラフ文法 [36, 19, 6] を用いるアプローチもある．

実用的なソフトウェア開発に適用可能なモデルコンパイラの作成を実現する場合，テキストの処理は不

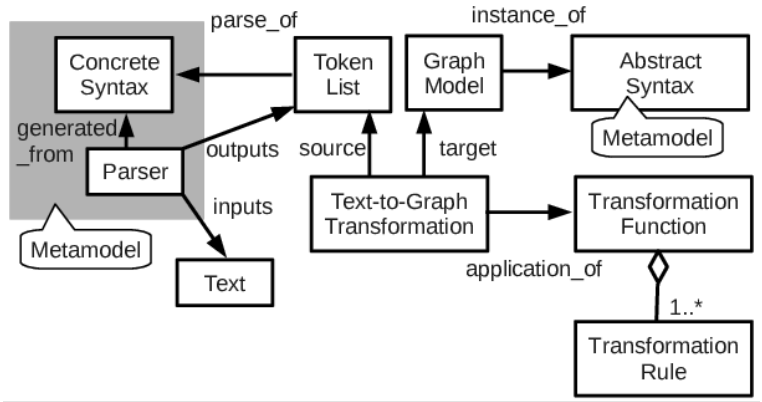


図 7.2: テキスト - グラフ変換

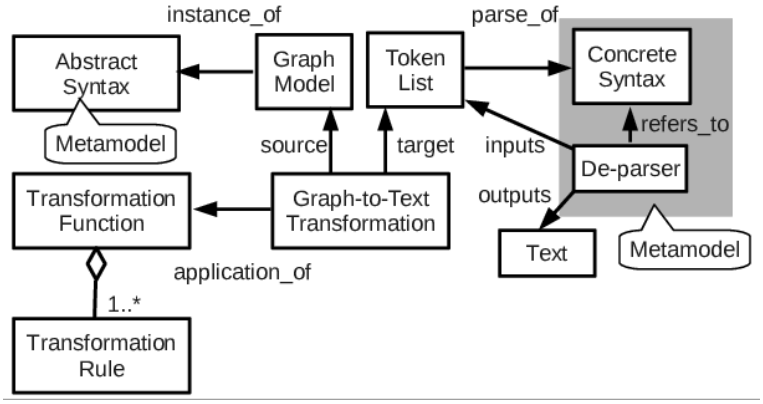


図 7.3: グラフ - テキスト変換系

可欠である．特定の表記形式でモデルを表現するプログラミング言語，仕様言語，DSL(Domain Specific Language) には多くの種類があることから，グラフモデルの背後にあるテキストを解析して変換する機能が必要である．

図 7.2 に示す構造は，図 7.1 の構造の特殊化であり，テキスト - グラフ変換を示す．入力テキストは具体的な構文から生成されたパーサによってトークンのリストに変換する．このトークンリストは，指定された変換規則に従ってターゲットグラフモデルの要素をインスタンス化するためのモデル変換の入力である．この場合，Parser および *Concrete Syntax* は図 7.1 の Metamodel の役割を果たし，関連には parse_of 関係を表す．トークンリスト (*Token List*) は，具体的な構文として指定されたメタモデルに基づいて生成される．一般に，モデル変換は，汎用プログラミング言語を使用して実装される．例えば，Yacc を使用する場合，トークンのリストをターゲットグラフモデル (多くの場合，抽象構文木) に変換するための各ルールは，C で書かれた一連のアクションによって指定される．

図 7.3 は，グラフからテキストへの変換の構造を示す．この場合，スキーマを保持しているデパーサ (*De-parser*) とともに具象構文がターゲットモデルのメタモデルの役割を果たす．

上述のように，モデルコンパイラは，グラフからグラフへ，テキストからグラフへ，およびグラフからテキストへの変換手続きを統合することによって実装される．特定のテキストの表記形式でグラフモデルを表現するプログラミング言語，仕様言語および DSL には多くの種類があることから，テキストを解析してグラフモデルに変換したり，グラフモデルからテキストを生成する機能は実用的なモデルコンパイラにとって不可欠である．

この 3 つのタイプの変換の間の差異を埋めるためにいくつかの研究が試みられてきた．MOFM2Text は，

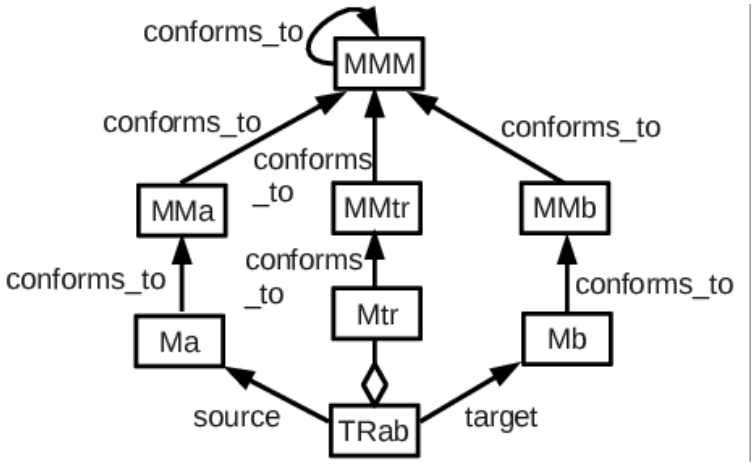


図 7.4: モデル変換パターン [33]

グラフ（MOF）モデルをテキスト表現に変換する方法に対処するために開発された．Fondement ら [25] は，具体的な構文として用いることが可能なメタモデルを提案した．Muller ら [47] は，モデル（グラフ）とコード（テキスト）の間に双方向変換を実装するツールを提案した．これらは，グラフからグラフ，テキストからグラフへのアプローチとグラフからテキストへのアプローチを統合するした実用的な MDE ツールに向けた有用な技術を提示している．

7.3 共通アーキテクチャの設計

前述のように，グラフとテキストの両方のモデルに柔軟に対応できるモデルコンパイラを作成するには，系統的なプロセスが必要である．ここでは，モデルコンパイラのための共通アーキテクチャの設計を示す．ソフトウェアアーキテクチャは，ソフトウェアの構造だけでなく，その構造が含意する開発プロセスも定義している [4]．

7.3.1 アーキテクチャ設計

本研究は，図 7.4 に示すよく知られているモデル変換パターン [33] を，共通アーキテクチャの基礎として用いる．このパターンはグラフからグラフへの変換のためのツールの抽象的な構造を示し，変換系 *TRab* はソースモデル *Ma* を分析することによってターゲットモデル *Mb* を作成する．変換手続きの仕様は，変換規則の組からなる変換モデル *Mtr* として定義される．モデル *Ma*，*Mb* および *Mtr* の抽象構文は，それぞれメタモデル *MMa*，*MMb* および *MMtr* で定義される．これらのメタモデルに共通する抽象構文はメタモデルモデル *MMM* で定義され，UML[8] による MDA[50, 45] の場合は MOF[52] に置き換えられる．

既存の MDE ツールとの相互運用性のために，図 7.4 のモデル変換パターンを拡張して共通アーキテクチャを設計する．テキスト形式で記述されたモデルを処理する手続きについては，7.2 章にのべた要件を考慮する必要がある．本研究は，グラフ処理のコンサーンとテキスト処理のコンサーンを識別した．これらは，ソースモデルとターゲットモデル（*Ma*，*Mb*）とメタモデル（*MMa*，*MMb*）に横断する．

図 7.5 は，モデルコンパイラのアーキテクチャを示す．開発者は，ソースモデルとターゲットモデルの表記法に基づいて，テキスト処理とグラフ処理に 2 つのコンサーンを組み合わなければならない．アスペクト指向技術を導入し，既存の技術とツールをこれらに柔軟に適用可能とした．

このアーキテクチャは 4 つのアスペクトモジュールによって構成される．*Parsing* および *De-parsing* アスペクトはテキスト処理の問題に対応し，*Graph analysis* および *Graph construction* アスペクトはグラフ処理の問題に対応する．

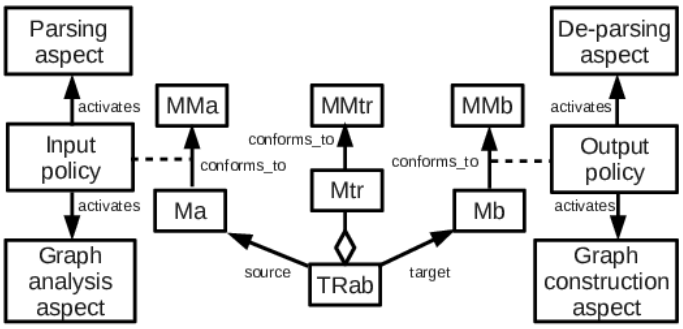


図 7.5: 共通アーキテクチャ

アスペクトの設計には，この横断的コンサーンの分離を目的 (インテント) として，(ベース) パターンを導出する．PBR パターンのコンポーネントとベースパターン導出のために与える役割の関係を表 7.1 に示す．このベースパターンを適用し，コンポーネントの振舞いを具体化することでアーキテクチャを設計する．

表 7.1: PBR パターンのコンポーネントに与える役割

コンサーン	PBR パターンのコンポーネント	役割
Parsing	Policy	InputPolicy
	Factory	ParserFactory
	Aspect Object	Parser
Deparsing	Policy	OutputPolicy
	Factory	DeparserFactory
	Aspect Object	Deparser
Graph Analysis	Policy	InputPolicy
	Factory	GraphAnalyserFactory
	Aspect	GraphAnalyser
	Object	
Graph Construction	Policy	OutputPolicy
	Factory	GraphConstructorFactory
	Aspect Object	GraphConstructor

図 7.5 の関連クラス（関連につながる点線のクラス）は，関連するオブジェクトの間の織込み関係を示す．関連クラス *Input policy* は，ソースモデルを入力しながら織込みポリシーをカプセル化する．このクラスのオブジェクトは，*Ma* から *MMa* へのメタモデル検索メッセージを奪い，メタモデルの表記形式（BNF などの具体的な構文）またはグラフ（MOF などの抽象構文）に応じて，*Parsing* または *Graph analysis* のいずれかのアスペクトを活性化または織込む．*Output policy* オブジェクトは，ターゲットモデルを出力している間に *Mb* から *MMb* へのメッセージを奪い，これに応じて *De-parsing* または *Graph construction* のアスペクトを織り込む．

図 7.6 は，より詳細な構造を示す．*Parsing* と *Graph analysis* の各アスペクトによって構成される．ソース

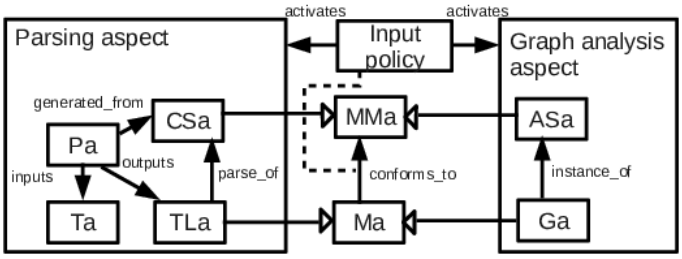


図 7.6: *Parsing* と *Graph analysis* アスペクト

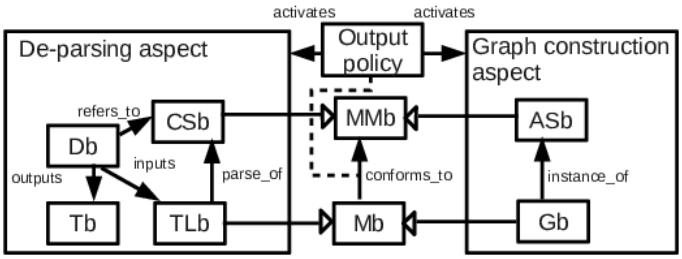


図 7.7: *De-parsing* と *Graph construction* アスペクト

モデル *Ma* が具体的な構文に従ったテキスト形式で記述されている場合，*Parsing* アスペクトは *Input policy* によって活性化される．ソースモデルの具体的な構文 *CSa* が与えられると，入力テキスト *Ta* をトークン (*TLa*) のリストに処理するためのパーサが生成される．その後，*Mtr* として指定されたモデル変換規則を *TLa* に適用して，ターゲットモデル *Mb* を生成する．*Ma* をグラフ形式で記述した場合，*Model analysis* アスペクトが活性化される．*Ma* の各要素が *MMb* で定義された関連する型のインスタンス化であることを確認した後，モデル変換ルール *Mtr* を適用して *Mb* を生成する。

図 7.7 は，*De-parsing* と *Graph construction* の 2 つのアスペクトを示す．ターゲットモデル *Mb* が具象構文に基づくテキスト形式であると予想される場合，*De-parser* アスペクトは *Output policy* によって活性化される．ターゲットモデルの具象構文 *CSb* を参照し，*De-parser* は，モデル変換（*TRab*）によって生成されたトークン *TLb* のリストを処理して，出力テキスト *Tb* を生成する．*Mb* がグラフ形式の場合，その要素はメタモデル *MMb* に基づいて作成される．

7.3.2 変換系の分類

本研究のアーキテクチャには 3 つの種類の変換ツールに分類できる．解析系，逆解析系およびグラフ変換系に分類できる．図 7.6 の *Pa* は解析系であり，逆解析系は図 7.7 の *Db* である．グラフ変換ツールのインスタンスは，図 7.5 の *TRab* として実現される．

解析系は，再帰的降下アルゴリズム，プッシュダウンオートマトン，有限状態オートマトンなどの構文解析技術に基づいて実現される．グラフ走査のアルゴリズムは，逆解析系の実装の基礎となる．グラフ変換ツールの開発には，サブグラフの同型性の問題がある．

7.3.3 メタモデルコンパイラへのアーキテクチャの適用

本研究の共通アーキテクチャーは，モデルコンパイラのメタレベルアーキテクチャとして用いることもできる．共通のアーキテクチャをメタモデルコンパイラの実装に適用することができる．アーキテクチャ自体は，アーキテクチャ内の変換系コンポーネントを生成するためのアーキテクチャとなる．

前節の任意の種類の変換系のメタモデルコンパイラは，テキストを読み書きする．入力テキストは，文脈自

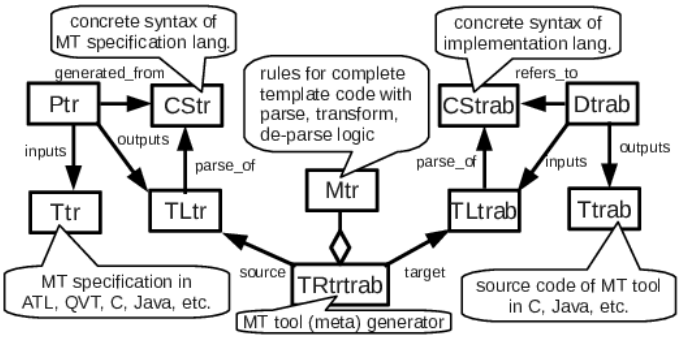


図 7.8: メタモデルコンパイラのためのアーキテクチャ

由文法またはグラフ文法の構文で記述される．本研究は一連のテキスト - グラフ，グラフ - グラフ，およびグラフ - テキスト変換系としてモデルコンパイラを設計することができる．

図 7.8 はモデルコンパイラのアーキテクチャである．図の *Parser* は、入力テキストが *Grammar* に適合しているかどうかを検査する．*Tansformation* は，プッシュダウンオートマトン，グラフ走査コンポーネント，抽象構文のサブグラフ同型を検査するコンポーネントなどである．*De-parser* は，抽象構文ツリーに対応する具象構文のテキストに変換する．*Tansformation* コンポーネントは，文法によって異なる．すなわち，文法は，生成される変換ツールの種類を指定する．

7.4 考察

共通のアーキテクチャの有用性を考察するために，テキストからグラフ，グラフからグラフ，グラフからテキストへの変換系を設計し，実装した．結果として，次のことがわかった．Text-to-Graph 変換系は，UNIX-yacc のような構文指向のプログラムジェネレータであることを示すことができる．グラフ - グラフ変換系にグラフ文法を適用した．サブツリー同型問題の DP アルゴリズムは，グラフを扱うために適用できる．グラフからグラフへの変換の場合，yacc のようにアクション記述の構文を定義する．グラフからテキストへの変換ツールは，解析しないスキームを持つグラフを走査するコンポーネントとして設計される．

7.4.1 テキスト変換系の事例

TeX では，HTML から HTML へ変換するテキスト変換系を実装した．変換系は，HTML-AST（抽象構文木）変換系，AST-AST 変換系，および AST-TeX 変換系で構成される．HTML-AST 変換系は，HTML 形式で記述されたテキストを読み取り，HTML の AST を出力する．AST-AST 変換系は，HTML-AST を TeX-AST に変換する．AST は，リスト項目の記述の表現が異なる．TeX のテキストは，AST-TeX コンバータによって生成される．図 7.9 はこのテキスト変換系を表している．

この実装を通じて，共通のアーキテクチャに基づいてモデルコンパイラを設計し実装することができることを確認できた．さらに，モデルコンパイラのモデルコンパイラも，アーキテクチャに基づいて設計・実装できると考えている．

7.5 まとめ

本研究では，モデルコンパイラのための共通のソフトウェアアーキテクチャを提案した．このアーキテクチャは，既存のモデルコンパイラから抽象化することにより，一般化された共通の構造を定義した．このアーキテクチャは，一般に用いられているモデル変換デザインパターンに基づいており，グラフからグラフ，テキストからグラフ，グラフからテキストへのモデル変換の構造と開発プロセスを説明可能なものとして定義した．

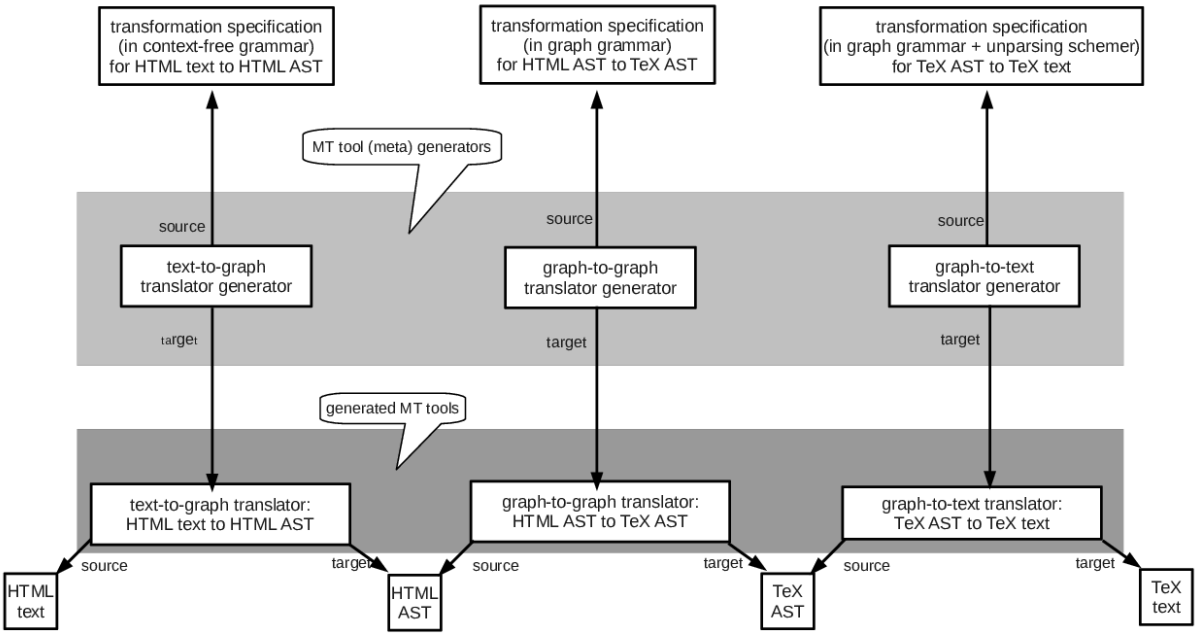


図 7.9: HTML から Tex へのテキスト変換系

アスペクト指向の概念を導入し，グラフからテキストへの変換および/またはグラフからテキストへの変換を，グラフからグラフへの変換に柔軟に織込むアーキテクチャを設計した．このアーキテクチャはメタレベルのモデルコンパイラにも適用可能であることも示した．

インタラクティブソフトウェアのモデルコンパイラを作成する事例を通じて，アーキテクチャの有用性と実用性を示した．

今後の課題は，提案したアーキテクチャに基づいてツールフレームワークを開発することである．アーキテクチャの関連コンポーネントに適用可能な技術の中から選択肢を提示するツールは，モデルコンパイラの開発者にとって有用であると考えた．アーキテクチャに基づいたモデルコンパイラのためのメタモデルコンパイラの開発は重要な課題である．メタモデルコンパイラはモデルコンパイラの開発の生産性を向上させると考えている．

第 8 章

IoT システムのためのソフトウェアアーキテクチャ

IoT システムは，組込みシステムとサービスを連携させて実現する．モバイル計算が実用化され，組込みシステムは移動体として設計・実現される．組込みシステムやサービスは，それぞれ外部環境に応じてその振舞いを変化させるコンテキストアウェアで実現される．本研究では，組込みシステムとサービスの間での協調は，メタコンテキストによって変化すると考える．本稿では，位置仮想化およびコンテキストを横断的コンサーンとして統一的に扱う IoT システムのためのアスペクト指向アーキテクチャを設計し，その有用性について議論する．自己適応のためのアーキテクチャパターンとして PBR パターンを定義した．IoT システムに対して，このパターンを適用することでコンテキストアウェアを実現した．これにより，アーキテクチャとアプリケーションの設計およびコードの理解，変更が容易になるだけでなく，ライブラリやミドルウェアを，大きな粒度で変更する枠組みが提供できた．PBR パターンを自己反映的に適用することで，メタコンテキストによる組込みシステムとサービスの間の協調を実現した．分割統治的に構造を整理したことにより，変更を行なうさいにその該当箇所が局所化され，柔軟に対応可能となった．

8.1 IoT システムの開発の現状および問題点と解決策

近年，ICT の発展に伴い，組込み機器をネットワークに接続し，相互通信を行なう Internet of Things(以下，IoT)[3] が普及してきた．また，モバイル計算が実用化され，組込みシステムを移動体として設計・実現される．IoT(Internet of Things) システムでは，組込みシステムの一部をサービスとして実現する．サービスにアクセスするために組込みシステム内に位置仮想化に関連するメッセージ通信が実現される．

IoT を用いることで，必要な情報をリアルタイムで取得することが可能となる．ユビキタス計算 [65] やパーベイシブ計算 [54] と同じ概念である．本研究では，アーキテクチャ設計にあたり前提として IoT の参照アーキテクチャに準拠したものとする．現在，様々な IoT の参照アーキテクチャが提案されている [5][23]．IoT の参照アーキテクチャ [7] は次の 4 層からなる (図 8.1)．

1. Data Center, Cloud Layer
   データセンタが置かれ，このサーバ上にサービスを配置する．詳細な処理が必要なデータ，即時応答の必要がないデータについて，より強力にデータを分析，管理，保管が可能な物理データセンタやクラウドベースのシステムに送信される．送信から応答まで時間がかかるものの、詳細な分析の実行だけでなく，ほかのデータを組み合わせた分析や処理が可能となる．
2. Core Network Layer
   IP ネットワークなど，複数のサブネットワーク間でデータを送信，交換する．コアネットワークのセキュリティサービスは，IoT システム全体の以下の脅威からの保護を実現する．
   Man-in-the-middle(MITM)：特定の 2 者間の通信に介在し，送信者と受信者になりすまし，盗聴

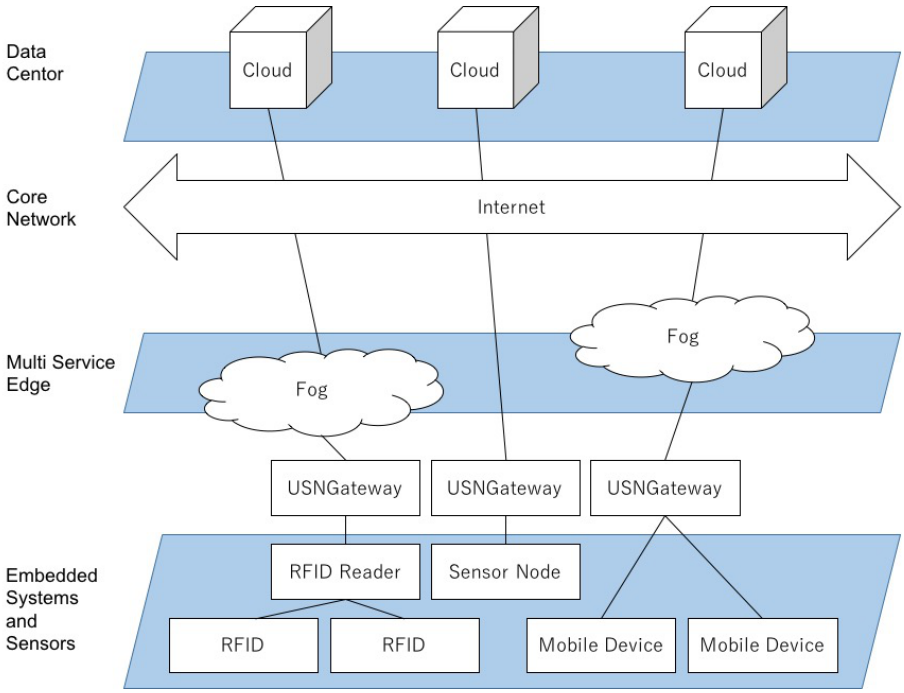


図 8.1: IoT の参照アーキテクチャ

および，偽装を行なう．
スプーニング：送信元の情報を偽装し，成りすましによって特定のエンドポイントに悪意のあるメッセージを送信する．
通信の改竄：中継されているデータの変更．
リプレイ攻撃：有効なデータを盗聴し，これを再送信することにより，身元を成りすまして不正なアクセスを行なう．

3. Multi Service Edge Layer
   組込みシステムやセンサなどをコアネットワークにアクセス可能とするために，この層に配置されるゲートウェイによってコアネットワークとの橋渡しを行なう．有線と無線による接続をサポートし，様々なエンドポイントに対応するために，Zigbee や，IEEE 802.11，3G，4G 等のプロトコルをサポートする．一般に，センサが存在する下位の層の構成要素に物理的に近い場所に配置される．データセンタへ送るデータの前処理の実行や，高い即応性が求められる処理については，この層に配置されるサービスによって行なわれる．フォグはこの層に対応する．
4. Embedded Systems and Sensors Layer
   組込みシステム，センサ，アクチュエータなどのデバイスが存在する．これらは，一般にコアネットワークにアクセスする機能を持たない．センサで可能な処理は，そのデバイスの処理能力によって制限される．この制限の下，即応性の高い処理は，エッジ層の最も物理的に近いサービスでデータを処理することで実現される．

近年，組込みシステムおよびサービスを，自己適応的に設計・実現する試みが行われている [31, 16]．自己適応は，外部環境を反映するシステムの内部状態をコンテキストとし，このコンテキストに応じた動的再構成で実現できる．コンテキストの内容を考察し，組込みシステムとサービスの協調においては，メタコンテキストの影響を受けてそれらの間の協調は変化するとの認識に至った (以下，コンテキスト協調と呼ぶ)．オブジェクト指向をコアコンサーンとし，位置仮想化およびコンテキストを含む横断的コンサーンについて適切なモ

ジュール化を行なうことが重要となる．

本研究の目的は，アーキテクチャ中心開発の基盤となる IoT システムのためのアーキテクチャを設計することである．この目的を達成するために，横断的コンサーンを分離した IoT システムのためのアスペクト指向アーキテクチャを設計する．

アーキテクチャ設計にあたり，自己適応のためのアーキテクチャパターンとして PBR パターンを定義した．PBR パターンを用いて位置仮想化およびコンテキスト等の横断的コンサーンを統一的に扱う．PBR パターンを適用することで，アーキテクチャとコードの理解，変更が容易になるだけでなく，ライブラリやミドルウェアを再利用する枠組みが提供できた．

コンテキスト協調については，自己反映的に PBR パターンを適用して実現した．コンテキストに関連する記述を分割統治的に行なうことができ，記述が簡便になる．コンテキスト協調させるメタコンテキストにポリシーを記述し，このメタコンテキストにあわせてコンテキストと振舞いの組を再構成する．これは手続き指向実現において，場合分けを複数の関数で記述することに相当する．すなわち，メタコンテキストを用いない記述は，case 文一文で全ての組み合せを記述することに相当し，複雑で理解しづらいものとなる．特定のコンテキストに応じた振舞いの変更のさいには，その変更の該当箇所が，特定のメタコンテキストにおけるコンテキストと振舞いの組に局所化された．また，その構造はベースとメタレベルで同じであることから，理解が容易である．

本論文で提案するアーキテクチャは，組込みシステムのためのアーキテクチャ [67] を改版し，IoT システムのためのアーキテクチャとして拡張したものである．

8.2 アーキテクチャの設計

OASIS のアーキテクチャの定義 [42] が一般的なアーキテクチャの設計および運用の枠組みを定義しているとの認識に立ち，この定義に基づいてアーキテクチャについて議論する．

8.2.1 IoT システムのための参照モデル

IoT システムのための参照モデル [24] は，一般に広く受け入れられていると考え，本研究でもこれを採用した．この参照モデルでは，ドメインモデル，情報モデル，機能モデル，通信モデル，セキュリティモデルを記述している．機能モデルは，ドメインモデルおよび情報モデルで整理された概念や情報を実現するための機能グループを整理している．本研究では，アーキテクチャ設計および実現の観点から，この機能モデルを参照アーキテクチャとして採用する．この機能モデルを図 8.2 に示す．要約すると，デバイス (*Device*) は，組込みシステムであり，センサやアクチュエータである．検知された値は，実世界を反映するオブジェクトモデルとして仮想化される (*VirtualEntity*)．IoT サービス (*IoTService*) とサービス編成 (*ServiceOrganization*) は，IoT プロセス管理 (*IoTProcessManagement*) によって協調が実現される．隣接する階層間でメッセージの授受があることを示している．

8.2.2 参照アーキテクチャ

位置仮想化，コンテキスト，メタコンテキストを含む横断的コンサーンを分離し，これらを統一的に扱うアスペクト指向参照アーキテクチャを設計した．

IoT システムの横断的コンサーン

IoT システムは，参照モデルに示されるように，組込みシステムとサービスが連携して実現されるものとして捉え，それぞれの横断的コンサーンを整理する．IoT システムにおける横断的コンサーンについて考察し，次の 6 つの横断的コンサーンを識別した．

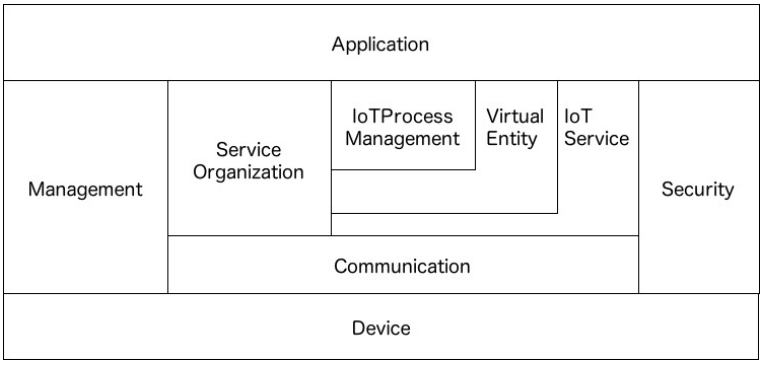


図 8.2: IoT システムのための参照モデル [24]

1. 並行コンサーン
2. コンテキストコンサーン
3. 実時間コンサーン
4. 耐故障コンサーン
5. 位置仮想化コンサーン
6. メタコンテキストコンサーン

これまでに，組込みシステムにおいてはオブジェクト指向をコアコンサーンとし，横断的コンサーンとして，並行，コンテキスト，実時間，耐故障コンサーンを識別した [67]．組込みシステムは，物理的な対象を制御し，これらは並行に動作するので，一般にソフトウェア上で並行に動作するオブジェクトの集合として定義されている [40]．実時間性，耐故障性などの非機能特性も設計し，実現しなければならない [1, 32]．移動体としての組込みシステムを考えた場合，コンテキストアウェアネスで実現することは有用である [18]．これらをまとめると，オブジェクト指向をコアコンサーンとし，並行コンサーン，コンテキストコンサーン，実時間コンサーン，耐故障コンサーンは横断的コンサーンとなる．

IoT システムでは組込みシステムのオブジェクト群の一部をサービスとして実現する．組込みシステム内に位置仮想化に関連するメッセージ通信が実現される．この位置仮想化コンサーンは横断的コンサーンとなる．また，サービスについてもコンテキストアウェアで実現されることから，コンテキストコンサーンが横断する．

コンテキスト協調は，メタコンテキストコンサーンとして，コンテキストコンサーンに対して横断する．コンテキストアウェアで実現される組込みシステムとサービスの協調のための，メタコンテキストに応じた再構成に関する記述が横断する．

前述の横断的コンサーンおよびコンテキストコンサーンの分離を目的 (インテント) として，(ベース) パターンを導出する．PBR パターンのコンポーネントとベースパターン導出のために与える役割の関係を表 8.1 に示す．このベースパターンを適用し，コンポーネントの振舞いを具体化することでアーキテクチャを設計する．

参照アーキテクチャの概要

図 8.3 に参照アーキテクチャの概要を示す．並行，位置仮想化，コンテキスト，メタコンテキストコンサーンは IoT システム全体に横断し，実時間，耐故障コンサーンは，一部に横断する．

以降より，IoT システムの参照アーキテクチャを説明する．ここでは，位置仮想化，コンテキスト，メタコンテキストコンサーンについて議論したいので，並行，実時間，耐故障コンサーンについては，省略する．非機能特性に関する横断的コンサーンは，静的な再構成として PBR パターンを適用して設計する．

表 8.1: PBR パターンのコンポーネントに与える役割

コンサーン	PBR パターンの コンポーネント	役割
並行性	Policy	SchedulingPolicy
	Factory	Scheduler
	Aspect Object	Thread
実時間性	Policy	TimingPolicy
	Factory	TimerFactory
	Aspect Object	Timer
耐故障性	Policy	Acceptance Policy
	Factory	F.T. HW Factory
	Aspect Object	HW
コンテキスト	Policy	Context
	Factory	Behavior Activator
	Aspect Object	Behavior
メタコンテキスト	Policy	MetaContext
	Factory	Behavior Activator
	Aspect Object	Behavior Activator

オブジェクト指向

組込みシステムでは，並行に動作する物理的なハードウェアに対してオブジェクトを定義する．ハードウェア (*HW*) の集合として設計した静的構造と動的振舞いを図 8.4(a)，(b) に示す．ハードウェア (*HW*) は，参照モデルに示されたようにセンサ (*Sensor*) とアクチュエータ (*Actuator*) に分類されるので，これらを多相型として定義した．センサ (*Sensor*) とアクチュエータ (*Actuator*) 両方の性質を持つもの (*SensorActuator*) に対しては多重 is-a 関係を用いて定義した．これら原始ハードウェア (*Primitive HW*) と複数の原始ハードウェアから構成される複合ハードウェア (*Composite HW*) を多相型として定義した．

コンテキスト

PBR パターンを適用し，コンテキストに関連する記述を分離する．コンテキスト指向プログラミング言語にあるように，コンテキストとこれに応じた振舞い，振舞いを活性化する手続きを分離し，独立に変更できるようにする．静的構造と動的振舞いを図 8.5(a)，(b) に示す．PBR パターンを適用し，ポリシーをコンテキスト (*Context*)，ファクトリを振舞い活性化手続き (*BehaviorActivator*) とした．*HW* 間のメッセージ通信を横

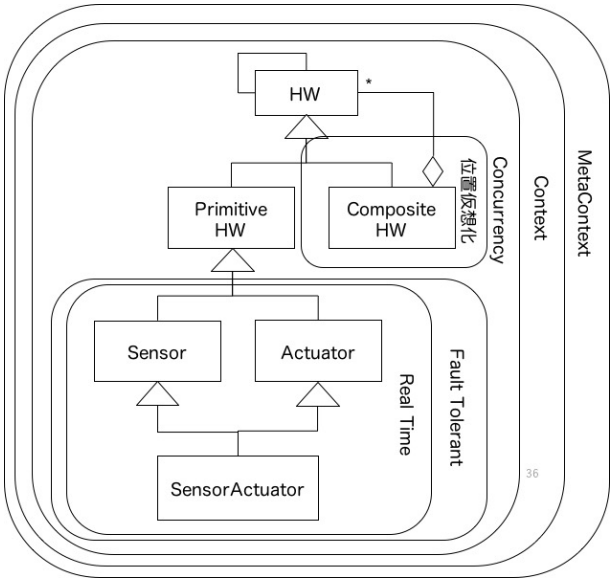


図 8.3: コアコンサーンと横断的コンサーンとの関係の概略

取りし，*Context* の状態を変化させる．*BehaviorActivator* は，*Context* の状態の変化に応じて，*HW* の振舞いを活性化させる．

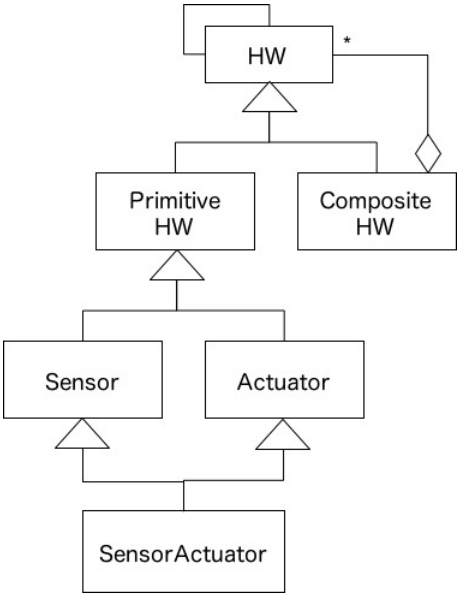
位置仮想化

IoT システムは, 図 8.3 に示したように一部の複合ハードウェアをサービスとして実現する．位置仮想化コンサーンによって規定されるサービスの集合の構造は，図 8.2 の参照モデルの階層に準ずる．サービスの集合の静的構造と動的振舞いを図 8.6(a)，(b) に示す．ハードウェアとサービスは，組込み機器 (*PrimitiveService*)，ゲートウェイ (*Gateway*)，フォグ上のサービス群 (*FogService*)，クラウド上のサービス 群 (*CloudService*) によって構成され，これらが協調するものとして定義した．これらのサービスに対しても図 8.3 に示したようにコンテキストコンサーンが横断する．

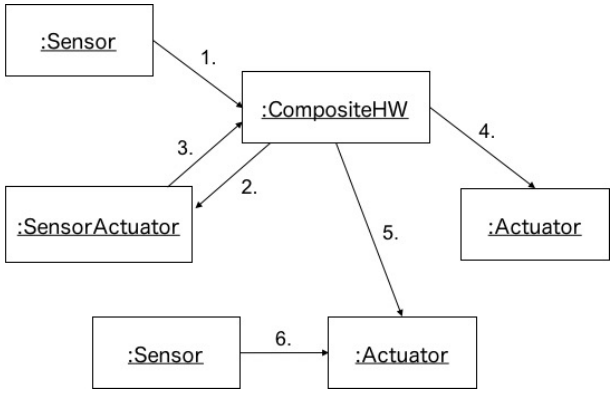
メタコンテキスト

PBR パターンを自己反映的に適用し，コンテキスト協調に関連する記述を分離する．コンテキストアウェアで実現される組込みシステムとサービスの間の協調は，互いのコンテキストに影響を受けて変化する．これはメタコンテキストに応じた，コンテキストと振舞いの関係の動的再構成である．自己反映的に適用することで，コンテキストコンサーンと同じ構造で，メタコンテキストと，ベースレベルのコンテキストと振舞いの組を再構成する手続きを分離し，独立して変更できるようする．この構造は，メタレベルとベースレベルで同じであることから，理解も容易である．静的構造と動的振舞いを図 8.7(a)，(b) に示す．*Object* 間のメッセージを横取りし，メタコンテキスト (*MetaContext*) の状態を変化させる．*BehaviorActivator* は，*MetaContext* の状態が変化したさいにコンテキストアスペクトの *BehaviorActivator* に定義されるコンテキストと振舞いの組を動的に再構成する．

以上，PBR パターンを用いて，それぞれの横断的コンサーンを統一的に記述できることを示した．



(a) 静的構造



(b) 動的振舞い

図 8.4: ハードウェア

8.2.3 具象アーキテクチャ

具象アーキテクチャは，実現技術を選択し，参照アーキテクチャの構造を詳細化したものである．実現技術とは，製品特有のモジュール構成法，プロトコル，および，適用するコード記述方法を指す．

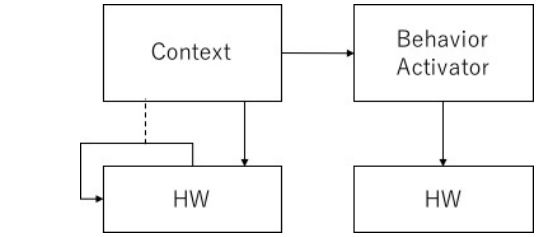
過去に開発した仮想店舗システムは，コンテキスト協調を行なう IoT システムであることから，これを設計するための具象アーキテクチャを説明する．

IoT システムに適用される実現技術

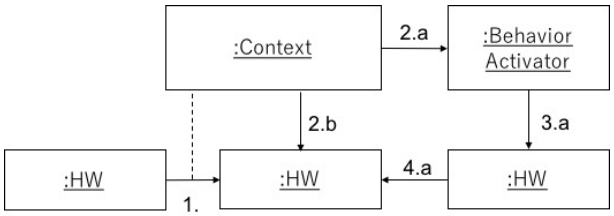
本研究で対象とする IoT システムの実現技術を整理すると以下の通りである．

- **モジュール構成法**

  モジュール構成法としては，ハードウェアの構成法と，自己適用に関連する構成法がある．ハードウェアは一般に状態遷移機械としてモデル化される [27]．自己適用技術に関連する構成法は，C2[63]，Weaves[30] がある．層モデルなど構造が明確なものを指向する場合は C2[63] を，オブ

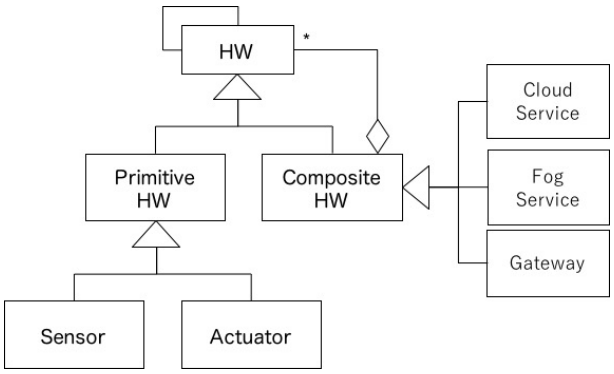


(a) 静的構造

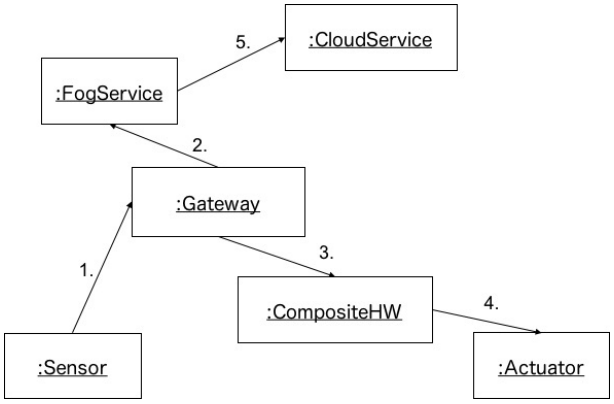


(b) 動的振舞い

図 8.5: コンテキストアウェアハードウェア

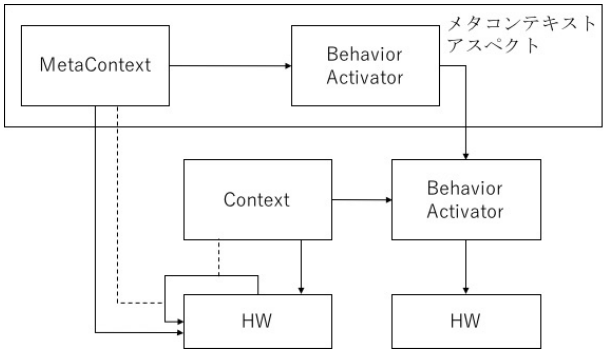


(a) 静的構造

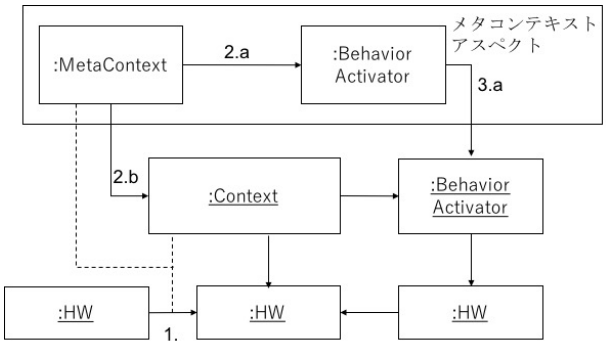


(b) 動的振舞い

図 8.6: 位置仮想化



(a) 静的構造



(b) 動的振舞い

図 8.7: コンテキスト協調

ジェクト指向スタイル [58] など構造の自由度が高い場合は Weaves[30] を用いるべきであることが知られている．

– **プロトコル**

アプリケーションフレームワークやライブラリが提供されているとして，アクセス方法が分かれば，アプリケーションを実装することができる．これをプロトコルの問題とする．並行プロセスの同期のためのプロトコルの代表的なものとして，Signal-Wait や PV 命令が選択可能である．耐故障処理のためのプロトコルの代表的なものとして，リカバリブロック，N バージョン，自己検査 (Self Checking Programming) が選択可能である [1]．コンポーネント間のメッセージ通信のプロトコルの代表的なものとして，リアクティブ (push 型) とポーリング (pull 型) が選択可能である．サービスとの通信におけるメッセージ形式のとして，URI によってメッセージを表現する形式や，パケットに格納するデータでメッセージを表現する形式があり，代表的なものとして REST メッセージや SOAP メッセージなどが挙げられる．

– **コード記述方法**

アスペクトの記述方法の代表的なものとして，Java を基本とした AspectJ がある．

仮想システムは，ハードウェアの構成が固定である．したがって，自己適用に関連するモジュール構成法として，C2 を選択した．

前述のとおり，ハードウェアは，一般に状態遷移機械としてモデル化されることから，これに従った．

実現技術は従属的である．すなわち，1 つの実現技術の選択により，別の実現技術の選択が決定する．例えば，プログラミング言語として，Java を選んだ場合，通常 Thread クラスライブラリを用いて並行処理を実現する．この Thread クラスを用いれば，必然的に同期のためのプロトコルは Sigal-Wait となる．仮想システムは，Java を用いて実現すること念頭に置いているので，Signal-Wait を選択した．状態遷移機械と従属的な

メッセージ通信のプロトコルとして，リアクティブ (push 通信) を選択した．サービスとの通信におけるメッセージ形式として，実行時効率を重視したいことから REST メッセージを選択した．

組込みシステムではメモリ制約があり，プログラムの冗長性を許容することができない．したがって，実行時のコードサイズが小さくなる実現技術を選択しなければならない．このことから，リカバリブロックを選択した．

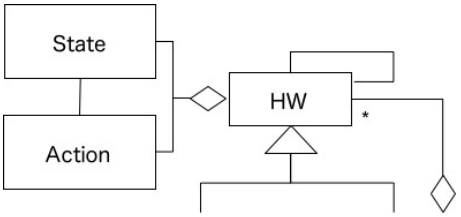
以上をまとめると，仮想システムでは，次の実現技術を選択した．

- **プログラミング言語：**Java
- **自己適用技術に関連する構成法：**C2
- **モジュールの構成法：**ハードウェアを状態遷移機械としてモデル化
- **並行性に関するプロトコル：**Signal-Wait
- **耐故障性に関するプロトコル：**リカバリブロック
- **サービスとの通信におけるメッセージ形式：**REST メッセージ

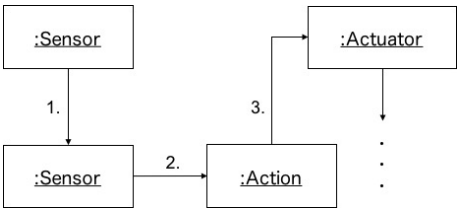
以降より，具象アーキテクチャを説明する．参照アーキテクチャと同様に，コンテキスト，メタコンテキストコンサーンについて議論したいので，並行，実時間，耐故障コンサーンについては，省略する．

オブジェクト指向

ソフトウェアの保守性を考慮し，ミーリ型状態遷移機械を導入する．この静的構造と動的振舞いを図 8.8(a), (b) に示す．ミーリ型状態遷移機械に対する変更として，状態およびアクション毎の変更や状態遷移の変更を独立して行なうことを目的として，状態とアクションを別のモジュールとして定義した (*State*, *Action*)．これによりそれぞれのモジュールの独立性を確保する．多相型については，図 8.4 と同じなのでここでは省略する．状態 (*State*) はイベントに応じてアクション (*Action*) を実行し，遷移後の状態 (*State*) を返すことで状態遷移を表現する．以降より，同様に状態遷移を導入し，適用されるコンポーネントにステレオタイプで <<STM>> と示す．



(a) 静的構造



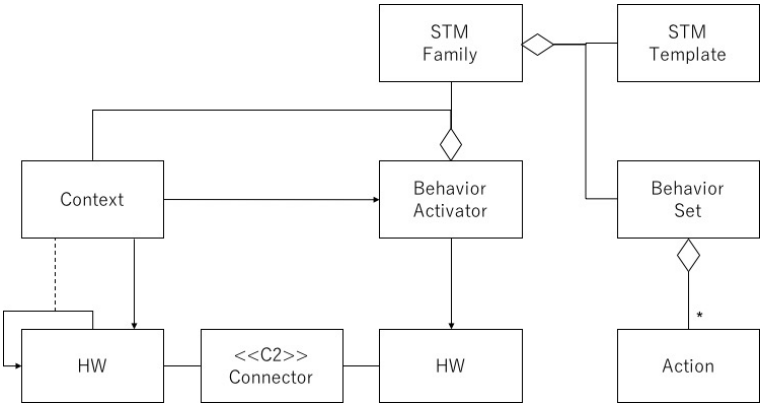
(b) 動的振舞い

図 8.8: ハードウェア

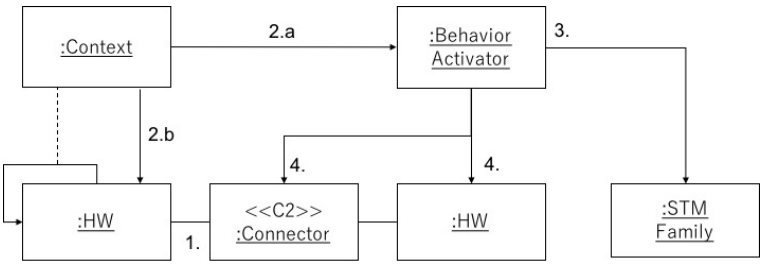
コンテキスト

コンテキストコンサーンに関する静的構造と動的振舞いは，コンテキストに依存して変化するアクションの変更を独立して行えるようにすることを目的として設計する．この静的構造と動的振舞いを図 8.9(a), (b)

に示す．ハードウェア (*HW*) を状態遷移機械として実現することから，この振舞いをコンテキストに応じて変更するために，コンテキストに依存したアクションの集合 (*BehaviorSet*) を持つ状態遷移機械ファミリ (*STMFamily*) を定義した．*Context* の状態に応じて *BehaviorActivator* は，*STMFamily* に *BehaviorSet* の持つ *Action* を組み合わせて *HW* を構築させる．これにより，差分となるアクションのみを独立して定義し，この組み合わせによってコンテキストに応じた構成を定義できるようになった．



(a) 静的構造



(b) 動的振舞い

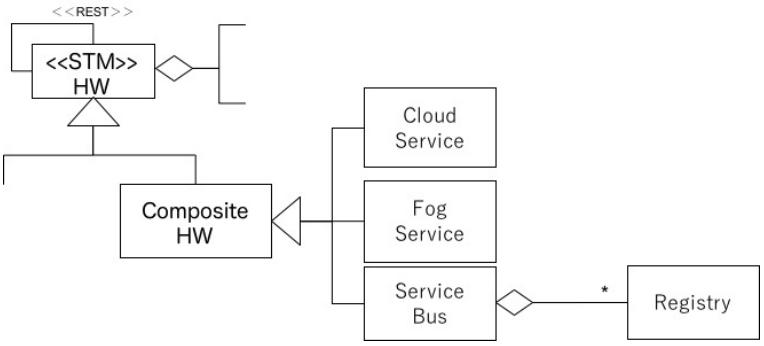
図 8.9: コンテキストアウェアハードウェア

位置仮想化

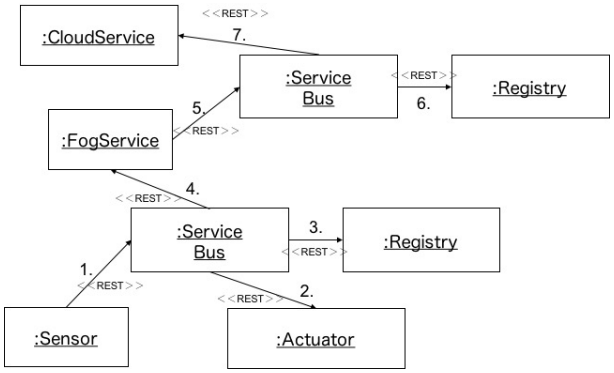
サービスアクセスとサービス構成管理の変更を独立して行なうことを目的として，ゲートウェイの構造としてサービスバス (*ServiceBus*) と，サービス特定のためのレジストリ (*Registry*) を導入した．この静的構造と動的振舞いを図 8.10(a)，(b) に示す．サービスに関連する多相型以外については，図 8.8 と同じなのでここでは省略する．

メタコンテキスト

コンテキストアスペクトと同じ構造を自己反映的に適用しする．この静的構造と動的振舞いを図 8.11(a), (b) に示す．コンテキストアスペクトの *BehaviorActivator* をメタコンテキストに依存したアクションの集合 (*BehaviorSet*) を持つ状態遷移機械ファミリ (*STMFamily*) によって構築する．*MetaContext* の状態に応じて *BehaviorActivator* は，*STMFamily* に *BehaviorSet* の持つ *Action* を組み合わせてコンテキストアスペクトの *BehaviorActivator* を構築させる．



(a) 静的構造



(b) 動的振舞い

図 8.10: 位置仮想化

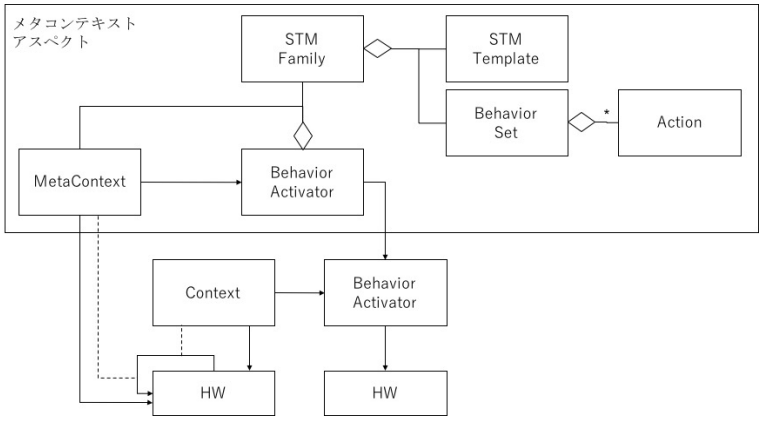
8.3 考察

本研究で調査した範囲では，コンテキスト指向技術を組込みシステムの設計・実装に応用する研究は行われているが [18]，メタコンテキストによるコンテキスト協調を実現する試みは行われていない．また，位置仮想化とコンテキストを含む横断的コンサーンを統一的に扱う試みは行われていない．PBR パターンを用いることによる利点は次の 3 通りである．

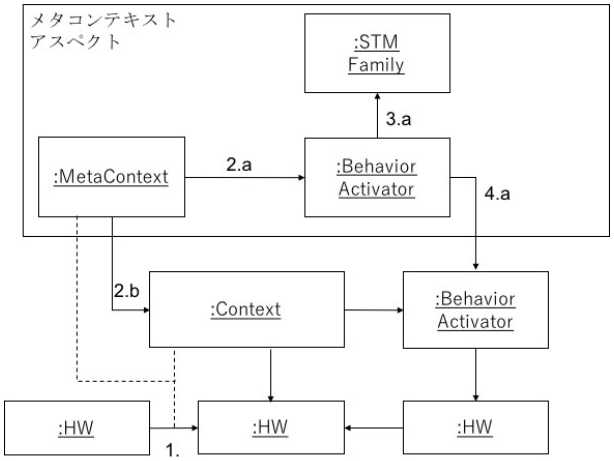
1. アーキテクチャレベルでは理解しやすい構造を提示
2. 変更の該当箇所の局所化が可能
3. 大きな粒度でのライブラリ等の再利用が可能

8.3.1 理解容易なアーキテクチャ

PBR パターンは，コアコンサーンと横断的コンサーンのモジュール分割のパターンであると同時に，分割されたモジュール群の協調に関連する記述を分離したものである．すなわち，コアコンサーンによって規定されるハードウェアの集合において，メッセージ通信の前後に横断的コンサーンによって規定されるモジュールへのメッセージ通信を付加する構造を示すものである．さらにすべての横断的コンサーンにおいても，協調に関連する記述や構造を標準化して定義した．PBR パターンだけを理解すれば，それぞれの横断的コンサーンの挙動を把握できる．本研究で設計した参照アーキテクチャは横断的コンサーンをコアコンサーンの構造と分離して互いに独立に記述し，その織込み方法を示したものとまとめられる．



(a) 静的構造



(b) 動的振舞い

図 8.11: コンテキスト協調

8.3.2 変更の該当箇所の局所化が可能

PBR パターンを自己反映的に適用することにより，複雑なコンテキストの組み合わせで記述されるコンテキスト協調の構造を整理し，変更するさいの該当箇所を局所化することが可能となる．

仮想店舗システムの事例では，スマートデバイスと仮想棚サービスが協調する．スマートデバイスは，位置情報に応じて仮想棚から店舗商品一覧または，全商品一覧を取得し，表示する．仮想棚から取得できる店舗商品一覧は，顧客の嗜好する商品の種類と店舗の商品在庫数に応じて変化する．これを素直に実現すると，顧客群が嗜好する商品の種類と商品在庫数の組み合わせをすべて記述することとなる．この組み合わせを case 文等で実現した場合，ネストは深くなり，特定の商品を顧客群が嗜好する場合の在庫に応じた商品の一覧を変更することは困難である．

本研究では，コンテキスト協調に関する変更を容易にすることを目的として，メタレベルとベースレベルからコンテキストに応じた振舞いを分割した．すなわち，メタコンテキストに応じてコンテキストに応じた振舞いを再構成する構造を定義した．具体的には，PBR パターンを用いてコンテキストコンサーンを分離し，さらにこれを自己反映的に適用することでメタコンテキストコンサーンを分離した．仮想店舗システムに適用したこの構造を図 8.12 に示す．スマートデバイスは位置情報，仮想棚は在庫をコンテキストとし，顧客群の嗜好情報をこれらのメタコンテキストとした．すなわち，特定の顧客群の嗜好に応じた，在庫と商品一覧を返す振

舞いとの組を定義した．コンテキストに応じた振舞いの変更のさいには，関連するメタコンテキストの時の場合について着目するだけでよく，素直に実現した場合と比較して変更の該当箇所を局所化できた．

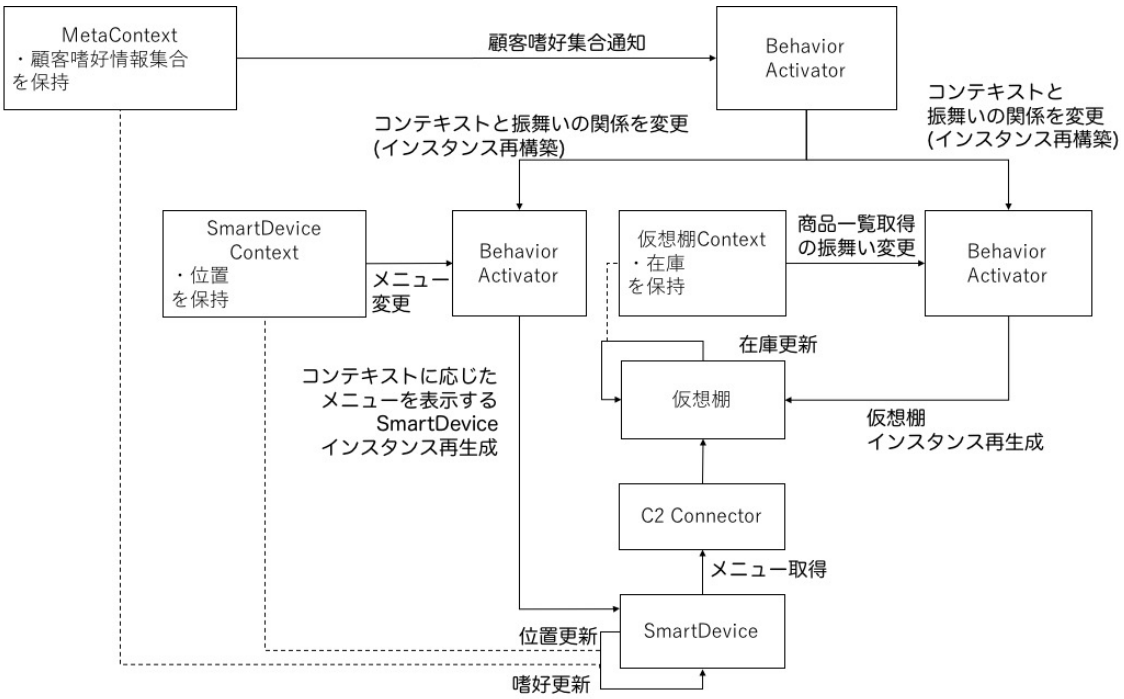


図 8.12: コンテキストアスペクト全体の静的構造

8.3.3 大きな粒度でのライブラリ等の再利用が可能

PBR パターンを用いることで，すべての横断的コンサーンについて独立して構造を導出でき，大きな粒度での再利用を可能となった．PBR パターンにより，アスペクトのコンポーネントは役割が明らかとなっており，その実現では，コンポーネント内でライブラリやミドルウェアを内包する．したがってこのコンポーネントの差替えによって，ライブラリやミドルウェアを再利用することができる．

8.4 まとめ

IoT システムは，移動体とサービスを連携させることで機能を実現する．クラウド [44] 上のサービスと連携することによる実時間性能の低下の問題を解決するためにフォグ [7] が提案されている．様々なサービスと連携することでアプリケーションドメインは拡大し，高機能化してきている．複数の非機能特性を考慮しながら実行時に変化するサービスや移動体に応じた振舞いを実現するには動的再構成の方法を考えることが重要である．移動体，フォグ，クラウドは相互に影響を受けて再構成されると我々は考えた．移動体にとって重視する非機能特性に応じてフォグの構成の変更が必要となる．実行時にアクセス可能なフォグやクラウド上のサービスは変化するので，移動体は非機能特性を考慮しながら，状況に応じたサービスを 1 つ選択する必要がある．IoT システムのためのアーキテクチャは，非機能特性を考慮しながら，構成要素間の関係を動的に再構成可能とするものとして設計する必要がある．

移動体およびフォグの動的再構成に関する要求を整理し，コンテキストに応じて Fog および組込みソフトウェアを動的に再構成するアーキテクチャを提案した．これにより，移動体およびフォグについて，非機能要求を充足した IoT アプリケーションの開発支援を実現する．IoT アプリケーションの構成要素間の関係の変化を動的に対応可能としたアーキテクチャを提案することにより，非機能要求を充足するために最適な構成を選

択可能とした．結果として，提案するアーキテクチャに基づいて実現することにより，コンテキスト間の協調によって移動体とフォグを再構成することが可能となった．アクセス可能な対象の中から，非機能特性を考慮して最適な対象 を選択することによる協調が実現される．

第 9 章

考察

本章では，次の 3 つの事項について考察する．

1. 問題解決に向けたアプローチの評価
2. PBR パターンによる技術課題の解決
3. PBR パターンを用いることによる利点

9.1 問題解決に向けたアプローチの評価

本節では，これまでの事例研究から，Salehie らの提案する局面の値すべてについてベースパターンにより取り扱い可能であることを考察する．局面とこれまでの 4 つの応用領域との関係を表 9.1 に示す．表の G とラベリングされている項目については，複数の事例を通して，特定の局面の値を網羅して記述し，結果を一般化して考察する．S とラベリングされている項目については，特定の事例において特定の局面の値を網羅して記述し，結果を一般化して考察する．本研究では，単純な記述を目指すことから，コストの局面については考慮しない．以下，それぞれの局面における現状と，PBR パターンによる試み，解決された内容を示す．

レイヤ

–**現状：** アプリケーションレベル，ミドルウェアレベル，基本ソフトウェアレベルにおいて，異なるアーキテクチャに基づく自己適応計算方式が提案されている．

–**課題：** 水平応用領域に独立な自己適応記述の可能性の考察する．

–**試み (G1-3)：** 提案パターンを適用する．

–**結果：** 単一のアーキテクチャパターン，12 種類のデザインパターン，2 種類のコーディングパターンで記述できることを確認した．

粒度

–**現状：** 粒度を固定し，自己適応のための言語，ミドルウェア，アプリケーションフレームワークが実現されている．

–**課題：** 疎粒度から細粒度までの自己適応記述の可能性の考察する．

–**試み (G4-6)：** 提案パターンを適用する．

–**結果：** 提案パターンを適用し，単一コンポーネント，コンポーネント群，サブシステムのどの粒度においても記述できることを確認した．

コスト 統一的な機構によって単純な記述になることを確認することを目的とするので考察の対象外とする．

動的/静的

–**現状：** 動的な再構成を実現するための言語，アプリケーションフレームワークが提案されている．アプリケーションフレームワークは言語依存であり，特定の言語で適応ポリシーを記述する．

–**課題：** 静的適応ならびに動的適応を統一的に設計するための構造を提案する．

–**試み (S7)：** デザインパターンを適用し，具象アーキテクチャを設計する．適応ポリシーを用いて動的/静的を実現するコーディングパターンを定義した．

–**結果：** 2 種類のデザインパターンで記述できることを確認した．

内部適応/外部適応

–**現状：** 専用言語においては，内部/外部はそれぞれ異なる言語により定義されている．デザインパターンとしては，内部適応はフックオペレーションパターンにより実現可能であるが，外部適応については提案されていない．

–**課題：** 内部適応ならびに外部適応を統一的に設計するための構造を提案する．

–**試み (G8-G9)：** 内部適応を行なう事例，外部適応を行なう事例にアーキテクチャパターンを適用し，参照アーキテクチャを設計する．

–**結果：** ベースレベル，メタレベルのどのレベルにおいても内部適応および外部適応を必要に応じて記述できることを確認した．

ソフトコンピューティング/ハードコンピューティング

–**現状：** ソフトコンピューティングのコーディングパターンの研究はなされているものの，ハードコンピューティングとソフトコンピューティングの統一的な記述に関する研究はなされていない．

–**課題：** ハードコンピューティングとソフトコンピューティングを統一的に設計するための構造を提案する．

–**試み (S10)：** ハードコンピューティングとソフトコンピューティングそれぞれについてデザインパターンを適用し具象アーキテクチャを設計，その実現においてもコーディングパターンを適用し実装する．

–**結果：** デザインパターンを用いることでハードコンピューティングとソフトコンピューティングが混在する場合において，それぞれを独立に記述できることを確認した．2 種類のコーディングパターンを用いることでハードコンピューティングとソフトコンピューティングのどちらについても記述できることを確認した．

オープン適応/クローズ適応

–**現状：** 安定性の観点からクローズ適応が一般に実現されている．アスペクト織込みのためのジョインポイントの動的追加可能なアプリケーションフレームワークが提案されている．

–**課題：** オープン適応とクローズ適応を統一的に設計するための構造を提案する．

–**試み (G11-12)：** アーキテクチャパターンを自己反映的に適用する．

–**結果：** 単一パターンによる単純な記述が得られた．

モデルベース適応/モデルフリー適応

–**現状：** 多くのものは，モデルベース適応を実現．モデルフリー適応の実現として，一般に特定の枠組みを与えている．

–**課題：** モデルベース適応とモデルフリー適応を 統一的に設計するための構造を提案する．

–**試み (G13-14)：**] アーキテクチャパターンを自己反映的に適用

–**結果：** 単一パターンによる単純な記述が得られた．

応用領域依存/一般化

–**現状：** DB など特定の垂直応用領域に特化したアプリケーションフレームワークや，設定により様々な自己適応が可能な汎用的なアプリケーションフレームワークが実現されている．

–**課題：** 垂直応用領域に独立な自己適応記述の可能性の確認をする．
–**試み (G15-18)：** 提案パターンを適用する．
–**結果：** G1-3 と同様の結果を得られた．

リアクティブ/プロアクティブ
–**現状：** リアクティブで実現されている．耐故障処理のための自己適応では，障害時の影響を小さくするために，プロアクティブなものが必要とされる．
–**課題：** リアクティブならびにプロアクティブを統一的な記述を提案する．
–**試み (S19)：** コーディングパターンを適用し，実装する．
–**結果：** 2 種類のコーディングパターンで記述できることを確認した．

継続的/適応的監視
–**現状：** 近年の多くの自己適応ソフトウェアは継続的モニタリングを実現している．実時間効率および資源効率を考慮すると，このモニタリングは好ましくない．
–**課題：** 継続的ならびに持続的モニタリングを統一的な構造を提案する．
–**試み (S20)：** デザインパターンを適用し，具象アーキテクチャを設計する．
–**結果：** 2 種類のデザインパターンで記述できることを確認した．

人間機械系
–**現状：** 人間機械系の自己適応ソフトウェア，自己適応の結果を人間に与える．
–**課題：** 人間機械系を対象とした自己適応記述の可能性を確認する．
–**試み (S21)：** インタラクティブシステムの事例に提案パターンを適用する．
–**結果：** 単一のアーキテクチャパターン，2 種類のデザインパターン，2 種類のコーディングパターンで記述できることを確認した．

適応結果の信頼性
–**課題：** 不確定要素を含む自己適応計算に対する開放的な構造を定義する．
–**試み (S22)：** インタラクティブシステムの事例に提案パターンを適用し，考察する．
–**結果：** オープン適応とソフトコンピューティングにより，最適化のための記述が得られた．

相互運用
–**現状：** 他の自己適応ソフトウェアと相互作用を行なうものは少ない．
–**課題：** コンテキストとアスペクトの相互作用に関する自己適応のための統一的な記述を定義する．
–**試み (S23)：** アーキテクチャパターンを自己反映的に適用する．
–**結果：** 相互運用のための単純な記述が得られた．

4 つの応用領域を通して，Salehie らの提案する局面の値を網羅して記述できた．このことから，PBR パターンから導出されたベースパターンによって，Salehie らの分類する自己適応計算方法それぞれについて取り扱い可能になったと言える．

PBR パターンは，代表的な自己適応ソフトウェアの参照モデルについて矛盾なく説明可能なものである．自己適応計算のための参照モデルとして代表的なものに，Kramer ら [37] は自己適応ソフトウェアの 3 層モデルを提案している．このモデルは次の 3 つの層からなる．

1. 目標管理層
2. 変更管理層
3. コンポーネント制御層

表 9.1: Salehie らの提案する局面と応用領域との関係

局面グループ	局面	Emb	I-Sys	Gen	IoT
適応対象	レベル	G1	G 1	G 2	G3
	粒度	G4	G 4	G 5	G6
	コスト				
適応方法	動的/静的	S7	S7		S7
	内部/外部	G8	G9	G8	G9
	ハードコンピューティング/ソフトコンピューティング		S10		
	オープン適応/クローズ適応	G11	G11	G11	G12
	モデルベース適応/モデルフリー適応	G13	G13	G14	G13
	応用領域特化/一般化	G15	G16	G17	G18
時制特性	リアクティブ/プロアクティブ	S19			S19
	継続的/適応的モニタリング	S20	S20		S20
適用結果の影響	人間機械系		S21		
	適応結果の信頼性		G22		
	相互運用性				S23

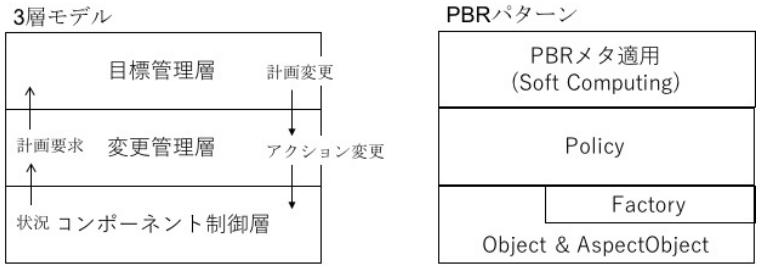


図 9.1: 3 層モデルおよび PBR パターンとの関係

最下層のコンポーネント制御層は，センサおよびアクチュエータの制御を行なう．現在の状況を変更管理層に通知し，変更管理層からのメッセージによりコンポーネントの作成，削除等を行なう．中間層の変更管理層は，状態の変化に反応し，計画に従ってコンポーネント制御層に変更メッセージを送る．最上位層の目標管理層は，想定していない変化が発生したさいに，計画の再構成を行なう．

この 3 層モデルは，自己反映的に適用した PBR パターンによって説明可能である．この 3 層モデルおよび PBR パターンとの関係を図 9.1 に示す．コンポーネント制御層は，コンポーネントとこの再構成の実行を行なう層であることから，PBR パターンでは，この層をさらに *AspectObject*，*Factory* に分割する．変更管理層は，変化に応じた再構成を計画することから，*Policy* に対応する．目標管理層は，想定しない変化に応じた *Policy* の再構成を行なうことから，PBR パターンを自己反映的に適用した層と考えられる．この層では，ソフトコンピューティングにより，再構成する *Policy* を動的に生成する．

9.2 PBR パターンによる技術課題の解決

PBR パターンを用いることによる 4 つの事例の技術課題の解決について考察する．アーキテクチャレベル，デザインおよびコードレベルについて述べる．

9.2.1 アーキテクチャレベル

5 章では，コンテキストおよび複数の非機能特性に関する横断的コンサーンを，統一的な構造で取り扱うアーキテクチャが設計できた．コンテキスト指向などの動的な自己適応の構造の標準化だけでなく，アスペクト指向の実現に代表される静的な自己適応のためのアスペクト間記述の構造を標準化できた．コンテキスト指向による動的な自己適応は，PBR パターンのコンポーネントをコンテキスト，振舞い活性化手続き，振舞いに具象化する．アスペクト指向による静的な自己適応は，織込みポリシー，織込み器，アスペクトに具象化する．結果として，自己適応の問題としてコンテキスト指向とアスペクト指向を再定義し，統一的な構造による簡便な記述を得られた．

6 章では，MVC とその派生のアーキテクチャ群を統一的に説明可能なアーキテクチャを定義できた．PBR パターンを用いることで，統一的な構造で定義された制御アスペクト，表示アスペクト，UI アスペクト，表示モデルアスペクトによって構成されるアスペクト指向アーキテクチャを定義した．このアスペクトのいくつかを織込むことで，MVC，AM-MVC，MVVM，PAC，MVP，HMVC アーキテクチャが導出可能となった．また，6.3.4 で考察したように，適応ポリシーコンポーネントをハードコンピューティングポリシーおよびソフトコンピューティングポリシーに具象化して実現することが可能となった．

7 章では，テキストからグラフへの変換系，グラフからグラフへの変換系，グラフからテキストへの変換系に分類されるモデルコンパイラのアーキテクチャを統一的な構造で定義した．PBR パターンを用いることで，統一的な構造で定義されたアスペクト群によって構成されるアスペクト指向アーキテクチャとして定義した．アスペクトの織込みによりそれぞれのモデルコンパイラが導出可能となった．このアーキテクチャを自己反映的に適用し，メタモデルコンパイラのアーキテクチャが定義できることを確認した．メタモデルコンパイラにより，モデルコンパイラの対象モデルの変更が可能となった．

8 章では，PBR パターンによって導出されたパターンを自己反映的に適用することにより，分割統治的にその構造が整理され，簡便な記述が得られることを確認した．複数のコンポーネントが自己適応のために参照するコンテキストは，それぞれのコンポーネントで保持しなければならないことから，アーキテクチャの構造を複雑にする．このようなコンテキストをメタコンテキストとして分離することにより，この複雑さが解消される．IoT システムの我々の例では，移動体とサービス間の複雑なコンテキストの組み合わせで記述されるコンテキスト協調の構造を整理した．

9.2.2 デザインおよびコードレベル

5 章では，PBR パターンからコンテキスト指向およびアスペクト指向のためのパターンとしてデザインパターンを導出・適用した．Factory に対し，Factory Method パターン等のデザインパターンの生成パターンを与えることで，言語機能として提供される静的な織込みの上で，動的な織込みを実現可能とした．これは，言語処理系のメカニズムの一部の使用者への開放と等価である．このデザインパターンによって定義されるコーディングパターンを用いて実現することができる．IoT システムの我々の例では，デザインパターンを用いた実現を試みた．

8 章では，自己相似的にパターンを適用した．自己相似的に適用しない素直な記述は，case 文一文で全てのコンテキストの組み合せを記述することに相当し，複雑で理解しづらく，変更が困難となる．自己相似的に適用すれば，このコンテキストの組み合わせの記述が分割統治的に整理され，特定のコンテキストに応じた振舞いの変更のさいには，その変更の該当箇所が局所化されることを確認した．

9.3 PBR パターンを用いることによる利点

PBR パターンを適用することで前述の技術課題が解決されることを確認できた．これによる利点として次の 3 つの事項が挙げられる．1. については，これまで考察してきたので，省略する．

1. Salehie らによって分類される自己適応計算方法を統一的に取り扱うことが可能
2. 実現コードの標準化
3. 技術転換すなわち，技術転換およびコード再利用が可能

参照アーキテクチャ設計から実装までのそれぞれの開発段階において PBR パターンを用いることができる．PBR パターンは，参照アーキテクチャ設計においては，アーキテクチャパターンを導出することができ，具象アーキテクチャ設計においては，デザインパターン，さらに，最も特殊化したものとして，アプリケーション実装においては，コーディングパターンを導出することができる．様々な開発段階における PBR パターンの役割と利点を図 9.2 に示す．

既存のアーキテクチャは，具象化されたベースパターンを適用して設計したものとして位置付けられる．既存のアーキテクチャの実現としてアプリケーションフレームワークやライブラリが定義されている．PBR パターンのコンポーネント群の実現としてのアプリケーションフレームワークは抽象化すれば同じ役割を持つことからこれらの間で置換が可能である．単体コンポーネントの実現としてのライブラリ間についても同様に置換が可能である．

デザインパターンとしての PBR パターンは，メタデザインパターンとデザインパターンに分類できる．

メタデザインパターンとしての PBR パターンは，既存のデザインパターン [28] を要素とした構造を定義するデザインパターンである．メタコーディングパターンとしての PBR パターンは，既存のデザインパターンに定義されているコーディングパターンを組み合わせたものを導出することができる．

デザインパターンとしての PBR パターンは，前述のようにその構造から，コンテキスト指向やアスペクト指向のための記述パターンとして扱うことができる．パターンのコンポーネントと特定の言語要素との対応関係からコーディングパターンが導出される．

PBR パターンは，自己適応のための構造を分離し，適応ポリシーコンポーネントと適応実行コンポーネントを独立して定義する．このことから，コンポーネント群や単体コンポーネントの単位で，その実現としてのアプリケーションフレームワークやライブラリ等を再利用することができる．

以下，これら利点の詳細について述べる．

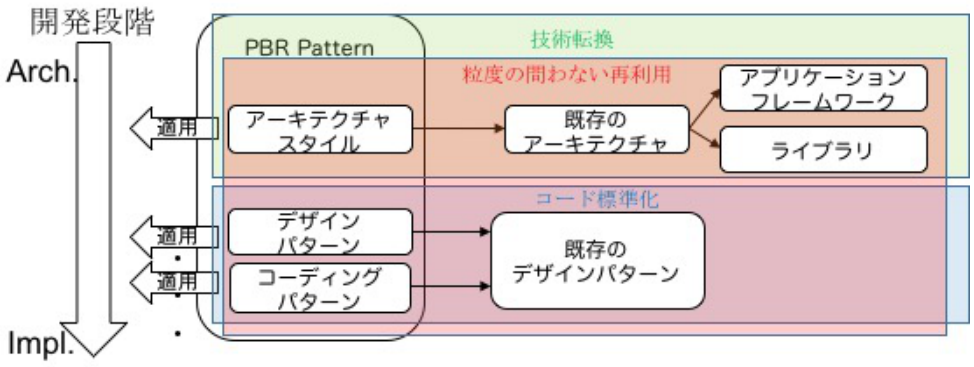


図 9.2: 様々な開発段階における PBR パターンの役割と利点

9.3.1 実現コードの標準化

PBR パターンと既存のデザインパターンとの関係から，標準化された実現コードの記述方法が定義できる．既存のデザインパターンは，コーディングパターンとしてコードの記述方法を定義している．このことから，アプリケーションに応じて詳細化に用いるデザインパターンを選択すれば，これを実現するためのコードの記述方法が決定できる．PBR パターンと既存のデザインパターン [28] との対応関係を表 9.2 に示す．例えば，適応ポリシーに与えるデザインパターンとして Mediator パターンや Command パターンを指定することができ，さらに，Observer を組み合わせることができる．Command と Observer を選択した場合，コンテキストコンポーネントにデータの変化を通知するメソッドとして notifyObserver メソッドが実装され，ポリシーコンポーネントに，データの変化が通知されるインターフェースとして doIt メソッドが実装されることが決定される．

PBR パターンは，特定の言語を用いた記述方法を定義する．アスペクト指向言語を用いて記述するさいには，アドバイス記述にこのパターンの構造が適用される．すなわち，適応ポリシーおよび生成手続きを独立に記述し，アドバイス記述からこれらへのメッセージ通信が記述される．コンテキスト指向言語を用いた場合の記述方法としては，適応ポリシーと振舞いバリエーションを独立して記述し，この適応ポリシーにより活性化させる振舞いバリエーションをビヘイビアアクティベータに指定するように記述される．

表 9.2: PBR パターンのコンポーネントと実現に適用可能なデザインパターンとの関係

PBR パターン のコンポーネント	実現に用いることのできる デザインパターン	説明
IAD	Template Method	メッセージング記述を分離
Policy	Mediator	ポリシー (Mediator) がコンテキストに応じて ファクトリを選択，起動
	Command	ポリシー群 (Command 群) がコンテキストに応じて ファクトリを選択，起動
	Observer	このパターンを組み合わせることで， リアクティブなポリシーの起動を実現
Factory	Abstract Factory	再構成対象のインスタンス生成手続きをまとめる
	Factory Method	再構成対象のインスタンスを生成
メタ記述	Prototype	メタ記述を分離
Aspect Object	Composite	このパターンによりモジュール化された 要素がコンテキストに応じて変化する
	Decorator	
	Template Method	
	State	
	Strategy	
なし	Singleton	再構成に適用することはできない
	Adapter	
	Facade	
	Flyweight	
	Chain of Responsibility	
	Iterator	

9.3.2 技術置換の枠組み

PBR パターンは，そのコンポーネント群および単体コンポーネントの実現としての，アプリケーションフレームワークやライブラリを，異なるバージョンや同じ目的を持った別のものに置換する枠組みを定義する．

自己適応のためのメタアーキテクチャパターンとして一般化された PBR パターンは，そのベースパターンを適用して設計されたアーキテクチャとして既存のアーキテクチャを定義できる．自己適応ソフトウェアのアーキテクチャの代表的なものとして，K-Component[20]，CASA[46]，Rainbow[12, 29] がある．これらを PBR パターンの実現として捉えることで，それぞれのアーキテクチャを統一的に説明することが可能となる．

K-Component は，Adaptation Contract，Architecture MetaModel，Model によって構成される．K-Component と PBR パターンとの対応関係を表 9.3 に示す．Architecture MetaModel は，Model のメタ記述である．Adaptation Contract は，条件と条件に応じた構成が記述される．Adaptation Contract に基づいて，特定の条件のとき，Architecture MetaModel を変更し，モデルにこの変更を反映させる．Adaptation Contract は適応ポリシーと再構成手続きがまとめて記述されていることから，PBR パターンにおける Policy および Factory に対応する．Architecture MetaModel は，PBR パターンにおける再構成対象の AspectObject に対応する．

CASA は，Adaptation Contract，Adaptive Middleware，Application によって構成される．PBR パターンと CASA との対応関係を表 9.4 に示す．Adaptation Contract は，条件と条件に応じた構成が記述される．Adaptive Middleware は，自己適応を実行する処理系であり，Application を変更する．Adaptation Contract は適応ポリシーと再構成手続きがまとめて記述されていることから，PBR パターンにおける Policy および Factory に対応する．Adaptive Middleware は自己適応を実行するための処理系であることから，Factory に対応する．Application は変更対象なので，Aspect Object に対応する．

Rainbow は，Architectural Model，Rules，Strategy によって構成される．PBR パターンと Rainbow との対応関係を表 9.5 に示す．Architectural Model は，再構成対象のメタモデルであり，コンテキスト情報をプロパティとして持つ．Rules には，再構成を行なう条件となる Architectural Model のプロパティとその時に選択される Strategy の関係が記述される．Strategy には，再構成のための Architectural Model のプロパティの変更処理が記述される．Architectural Model は，再構成対象であり，コンテキスト情報も持つことから，PBR パターンにおける，Aspect Object と Context Data に対応する．Rule には，再構成を行なう条件とその時に選択される再構成手続きとの関係が記述されることから，Policy に対応する．Strategy は，再構成手続きそのものなので，Factory に対応する．

表 9.3: K-Component[20] と PBR パターンとの関係

K-Component のコンポーネント	対応する PBR パターンのコンポーネント	再利用可能なもの
Adaptation Contract	Policy	ACDL(Adaptation Contract Description) の処理系
	Factory	
Architecture Meta Model	Aspect Object	

既存のアーキテクチャの実現として定義されているアプリケーションフレームワークやミドルウェアは PBR パターンに基づく実現として定義できる．したがって，PBR パターンの特定のコンポーネント群またはコンポーネントの実現間での相互置換が可能となる．例えば，PBR パターンのコンポーネント群の実現としての K-Component，CASA，Rainbow 間での置換が可能である．ファクトリコンポーネントによる適応処理の実現に Odyssey[49] や QuO[41] を用いることができ，このコンポーネントの実現単位で，これらライブラ

表 9.4: CASA[46] と PBR パターンとの関係

CASA のコンポーネント	対応する PBR パターンのコンポーネント	再利用可能なもの
Adaptation Contract	Policy	Policy 言語 (XML ベースの言語) の処理系
	Factory	
Adaptive Middleware		適応処理ミドルウェア Odyssey[49] QuO[41]
Application	Aspect Object	

表 9.5: Rainbow[12, 29] と PBR パターンとの関係

Rainbow のコンポーネント	対応する PBR パターンのコンポーネント	再利用可能なもの
Architectural Model	Context Data, Aspect Object	Model Manager(処理系)
Rules	Policy	Adaptation Manager (処理系)
Strategy	Factory	Strategy Manager (処理系)

リ間での置換が可能である.

9.3.3 再利用の枠組み

PBR パターンは，様々な抽象度における再利用のための枠組みを提供する.

メタアーキテクチャパターンとしての PBR パターンでは，そのベースパターンとして既存のアーキテクチャを再利用を支援する．PBR パターンのコンポーネントの役割から，既存のアーキテクチャのコンポーネントを理解を容易にする.

メタデザインパターンとしての PBR パターンでは，既存のデザインパターンの組み合わせを再利用可能とする．表 9.2 に示した適用可能なデザインパターンの組み合わせによる既存のデザインパターンの再利用が可能となる.

PBR パターンによって導出されるベースパターンのコンポーネント群の実現として，既存のアプリケーションフレームワークを位置付けることができるので，高い抽象度で再利用が可能である．PBR パターンは，アスペクトとして独立して定義することから，このコンポーネント群をまとめて再利用することが容易である.

ベースパターンのコンポーネントの実現として，既存のライブラリや既存のコーディングパターンを位置付けることができるので，このコンポーネントの抽象度での再利用が可能である．PBR パターンは，適応ポリシーコンポーネント，適応実行コンポーネントを独立して定義することから，このコンポーネント単位での再利用が容易である.

9.3.4 ソフトウェアプロセスの標準化

本研究では，PBR パターンを適用することによるソフトウェアプロセスについては定義していないが，PBR パターンの構造により以下のような標準化されたソフトウェアプロセスを定義および PLSE(Product Line

Software Engineering) 応用ができると考えている．これらについては，今後の課題とする．

PBR パターンの構造は，ソフトウェアプロセスとして，特定の状況に応じた振舞い，適応ポリシー，適応処理について考慮して詳細化することを定義している．

領域工学 (Domain Engineering) では，要求に基づき，コア資産としての参照アーキテクチャの選択または設計を行なう．開発環境 (DE)，実行時環境 (RE) を選択し，参照アーキテクチャに基づいて具象アーキテクチャを選択または設計する．その後，コア資産の中から，具象アーキテクチャの実現として利用できるコンポーネントを選択する．

参照アーキテクチャ設計では，メタアーキテクチャパターンとしての PBR パターンを用いて，アーキテクチャレベルの役割を与えて，自己適応計算の構造を定義する．例えば，5.2.3 の図 5.6 では，パタメータとして，*SchedulingPolicy*，*Scheduler*，*Thread* を与えて適用することで，並行性アスペクトを表現した．

具象アーキテクチャ設計段階では，メタデザインパターンまたはデザインパターンとしての PBR パターンを適用し，デザインレベルの役割を与えて，自己適応計算の構造を定義する．メタデザインパターンのコンポーネントと適用可能な既存のデザインパターンとの関係は，前述の表 9.2 に示している．

応用工学 (Application Engineering) では，具象アーキテクチャに基づきアプリケーション設計を行ない，領域工学で選択したコンポーネントを用いながら実装する．この既存のデザインパターンに定義されているコーディングパターンを用いて実装する．また，Rainbow などの既存の実現技術を用いた開発では，前述の PBR パターンとの対応関係に従い，特定の状況に応じた振舞い，適応ポリシーおよび適応処理について指定された言語で記述する．

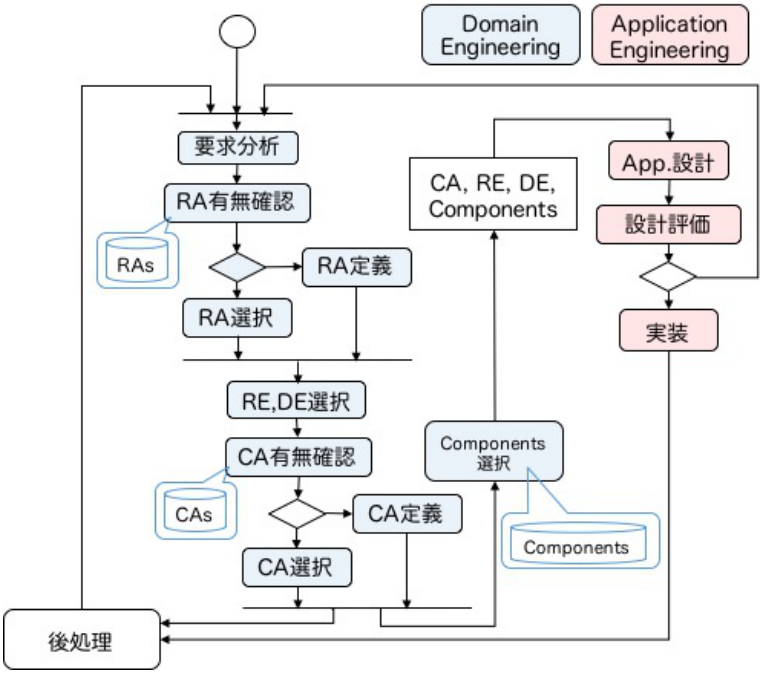


図 9.3: アーキテクチャ中心開発プロセス

第 10 章

結論

本章では，研究の成果をまとめ，最後に今後の課題について述べる．

10.1 まとめ

本研究では，自己適応のためのメタパターンとしての PBR パターンを定義し，この有用性を考察した．組込みシステム，インタラクティブシステム，IoT システム，コード生成系それぞれの応用領域の事例において，このメタパターンから導出されるベースパターンを適用し，考察した．

PBR パターンを用いることによる利点として次の 3 つの事項が挙げられる．

1. Salehie らによって分類される自己適応計算方法を統一的に取り扱うことが可能
2. 実現コードの標準化
3. 技術転換すなわち，技術転換およびコード再利用が可能

Salehie らによって分類されるすべての自己適応計算方法について，複数の応用領域の事例研究を通して，共通のメタパターンから導出されるベースパターンにより取り扱い可能であることを確認した．関連研究では，特定の応用領域や自己適応計算方法に依存したアーキテクチャおよびその実現が定義されている．PBR パターンを用いることで，アーキテクチャレベルからコードレベルまで統一的な構造で定義でき，矛盾のない一貫した記述および言語独立が実現できた．また，この統一的な取り扱いに基づき，技術転換，すなわち，置換および再利用の枠組みを実現できた．PBR パターンから導出されるパターンと既存のアーキテクチャとの関係から，既存のアーキテクチャの実現としてのアプリケーションフレームワーク，ライブラリ等の技術置換や粒度の大きな再利用が可能となった．新たなアプリケーションフレームワークやライブラリが提供されたさいにも，PBR パターンとの対応関係が定義できれば技術置換を行なうことが可能となる．PBR パターンは，他の既存研究の理解の支援や，その成果物を用いた開発の支援を実現するものとして位置付けられる．

10.2 今後の課題

PBR パターンの構造に基づくソフトウェアプロセスの定義および PLSE 応用の可能性を考察する．PBR パターンの構造に基づくソフトウェアプロセスは，計算方法や応用領域によらない標準化されたものである．このソフトウェアプロセスを定義することで，より自己適応ソフトウェアの作成の支援が可能となる．

アプリケーションフレームワークおよびライブラリ間のインターフェースの差異を吸収するための方法を提案する必要がある．PBR パターンは，コードレベルでは，共通のインターフェースを定義するものである．PBR パターンとの対応関係から，特定のライブラリに対して，類似するライブラリを特定することが可能である．アプリケーションフレームワークやライブラリのインターフェースは，それぞれ独自に定義されていることから，技術置換や再利用を容易にするためには，PBR パターンの定義する共通のインターフェースとの差異を吸収する方法を定義する必要がある．

ソフトコンピューティングポリシーが実現可能であることを実用的なシステムで確認する必要がある．本研究では，ソフトコンピューティングポリシーを実現していない．例えば，インタラクティブシステムでは，ユーザの操作履歴をコンテキストとし，適応ポリシーで学習することによってコンテキストからユーザの嗜好を特定し，ユーザの嗜好に合った画面を表示するように再構成することを可能とする．このことを実用的なシステムで実装し，確認する必要がある．

同じ役割を持つ自己適応技術間の実時間効率および資源効率関係を明らかにする必要がある．PBR パターンにより，技術置換および再利用が容易になった．実時間効率および資源効率と実現技術の関係が明らかとなっていないので，これらを考慮した実現技術の選択は，試行錯誤によって行なう必要がある．実時間効率および資源効率と実現技術の関係を定義する枠組みが定義できれば，選択が容易になるだけでなく，実現技術の自動選択や，PLSE の基礎として用いることが可能となる．これにより，PLSE 応用を実現するための基礎が定義できると考える．

参考文献

[1] Angelov, C., Marian, N., Sierszecki, K., and Ma, J.: Model-Based Design and Verification of Embedded Software, *Proc. of the 5th European Workshop on Research and Education in Mechatronics*, 2004.

[2] Appeltauer, M., Hirschfeld, R., Haupt, M., and Masuhara, H.: ContextJ: Context-Oriented Programming with Java, *Information and Media Technologies*, Vol. 6, No. 2(2011), pp. 399–419.

[3] Ashton, K.: That 'Internet of Things' Thing, *RFiD Journal*, Vol. 22, No. 7(2009), pp. 97–114.

[4] Bass, L.: *Software architecture in practice*, Pearson Education India, 2007.

[5] Bassi, A., Bauer, M., Fiedler, M., Kramp, T., Van Kranenburg, R., Lange, S., and Meissner, S.: *Enabling Things to Talk*, Springer-Verlag Berlin Heidelberg, 2013.

[6] Besova, G., Steenken, D., and Wehrheim, H.: Grammar-based model transformations, *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, IEEE, 2014, pp. 1601–1610.

[7] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S.: Fog computing and its role in the internet of things, *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, ACM, 2012, pp. 13–16.

[8] Booch, G.: *The unified modeling language user guide*, Pearson Education India, 2005.

[9] Cai, J., Kapila, R., and Pal, G.: HMVC: The layered pattern for developing strong client tiers, *Java World*, (2000), pp. 07–2000.

[10] Candea, G., Cutler, J., and Fox, A.: Improving availability with recursive microreboots: A soft-state system case study, *Performance Evaluation*, Vol. 56, No. 1(2004), pp. 213–248.

[11] Cetina, C., Giner, P., Fons, J., and Pelechano, V.: Autonomic computing through reuse of variability models at runtime: The case of smart homes, *Computer*, Vol. 42, No. 10(2009), pp. 37–43.

[12] Cheng, S.-W., Garlan, D., and Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system, *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2009*, IEEE Computer Society, 05 2009, pp. 132–141.

[13] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A.: Live migration of virtual machines, *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, USENIX Association, 2005, pp. 273–286.

[14] Clarke, S. and Baniassad, E.: *Aspect-oriented analysis and design*, Addison-Wesley Professional, 2005.

[15] Coutaz, J.: PAC, *ACM SIGCHI Bulletin*, Vol. 19, No. 2(1987), pp. 37–41.

[16] Cumby, C., Fano, A., Ghani, R., and Krema, M.: Building intelligent shopping assistants using individual consumer models, *Proceedings of the 10th international conference on Intelligent user interfaces*, ACM, 2005, pp. 323–325.

[17] Czarnecki, K. and Helsen, S.: Feature-based survey of model transformation approaches, *IBM Systems Journal*, Vol. 45, No. 3(2006), pp. 621–645.

[18] Daftari, A., Mehta, N., Bakre, S., and Sun, X.-H.: On Design Framework of Context Aware Embedded Systems, *Monterey workshop on software engineering for embedded systems: From requirements to implementation*, 2003.

[19] De la Parra, F.: Application of Graph Grammars to Model Transformations, Technical report, Tech. Report 2013-604, School of Computing Queen' s University Kingston, Canada, 2013.

[20] Dowling, J. and Cahill, V.: Self-managed decentralised systems using K-components and collaborative reinforcement learning, *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, ACM, 2004, pp. 39–43.

[21] Ekaterina, N.: 15 BENEFITS OF SOFTWARE ARCHITECTURE, https://www.linkedin.com pulse/15-benefits-software-architecture-ekaterina-novoseltseva/.

[22] Erradi, A., Maheshwari, P., and Tosic, V.: Policy-driven middleware for self-adaptation of web services compositions, *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Springer-Verlag New York, Inc., 2006, pp. 62–80.

[23] Evans, D.: A Reference Architecture for the Internet of Things, http://wso2.com, 2015.

[24] FhG, I., SAP, S. H., HSG, E. H., Jardak, C., CEA, A. O., Serbanati, A., SAP, M. T., and Walewski, J. W.: Internet of Things-Architecture IoT-A Deliverable D1. 3–Updated reference model for IoT v1. 5.

[25] Fondement, F. and Baar, T.: Making metamodels aware of concrete syntax, *ECMDA-FA*, Vol. 3748(2005), pp. 190–204.

[26] France, R. and Rumpe, B.: Model-driven development of complex software: A research roadmap, *2007 Future of Software Engineering*, IEEE Computer Society, 2007, pp. 37–54.

[27] Gajski, D. D., Abdi, S., Gerstlauer, A., and Schirner, G.: *Embedded System Design: Modeling, Synthesis and Verification*, Springer Science & Business Media, 2009.

[28] Gamma, E., Richard, H., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

[29] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure, *Computer*, Vol. 37, No. 10(2004), pp. 46–54.

[30] Gorlick, M. M. and Razouk, R. R.: Using Weaves for Software Construction and Analysis, *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, Los Alamitos, CA, USA, IEEE Computer Society Press, 1991, pp. 23–34.

[31] Hirschfeld, R., Costanza, P., and Nierstrasz, O.: Context-Oriented Programming, *Journal of Object Technology*, Vol. 7, No. 3(2008), pp. 125–151.

[32] Hsiung, P.-A., Lin, S.-W., Tseng, C.-H., Lee, T.-Y., Fu, J.-M., and See, W.-B.: VERTAF: An Application Framework for the Design and Verification of Embedded Real-Time Software, *IEEE Trans. Softw. Eng.*, Vol. 30, No. 10(2004), pp. 656–674.

[33] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I.: ATL: A model transformation tool, *Science of computer programming*, Vol. 72, No. 1(2008), pp. 31–39.

[34] Kephart, J. O. and Chess, D. M.: The Vision of Autonomic Computing, *Computer*, Vol. 36, No. 1(2003), pp. 41–50.

[35] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G.: An Overview of AspectJ, *European Conference on Object-Oriented Programming*, Springer, 2001, pp. 327–354.

[36] Kong, J., Zhang, K., Dong, J., and Song, G.: A graph grammar approach to software architecture verification and transformation, *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, IEEE, 2003, pp. 492–497.

[37] Kramer, J. and Magee, J.: Self-managed systems: an architectural challenge, *2007 Future of Software*

Engineering, IEEE Computer Society, 2007, pp. 259–268.

[38] Krasner, G. E., Pope, S. T., et al.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system, *Journal of object oriented programming*, Vol. 1, No. 3(1988), pp. 26–49.

[39] Lano, K. and Kolahdouz-Rahimi, S.: Model-transformation design patterns, *IEEE Transactions on Software Engineering*, Vol. 40, No. 12(2014), pp. 1224–1259.

[40] Lee, E. A.: Embedded software, *Advances in computers*, Vol. 56(2002), pp. 55–95.

[41] Loyall, J., Bakken, D., Schantz, R., Zinky, J., Karr, D., Vanegas, R., and Anderson, K.: QoS aspect languages and their runtime integration, Springer, 1998, pp. 303–318.

[42] MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R., and Hamilton, B. A.: Reference model for service oriented architecture 1.0, *OASIS standard*, Vol. 12(2006), pp. 18.

[43] Marcotte, E.: Responsive web design, 2010, http://alistapart. com/article/responsive-web-design, 2010.

[44] Mell, P. and Grance, T.: The NIST definition of cloud computing, (2011).

[45] Mellor, S. J.: *MDA distilled: principles of model-driven architecture*, Addison-Wesley Professional, 2004.

[46] Mukhija, A. and Glinz, M.: Runtime adaptation of applications through dynamic recomposition of components, *International Conference on Architecture of Computing Systems*, Springer, 2005, pp. 124–138.

[47] Muller, P.-A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., and Jézéquel, J.-M.: Model-driven analysis and synthesis of concrete syntax, *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 98–110.

[48] Nakajima, S.: Detecting feature interferences in PHP-based web applications, *Proc. 22nd ICSSEA*, (2010).

[49] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R.: Agile application-aware adaptation for mobility, *ACM SIGOPS Operating Systems Review*, Vol. 31, No. 5, ACM, 1997, pp. 276–287.

[50] ObjectManagementGroup: MDA — The Architecture of Choice for a Changing World, http://www.omg.org/MDA/.

[51] ObjectManagementGroup: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, http://www.omg.org/spec/QVT/.

[52] ObjectManagementGroup: OMG Meta Object Facility (MOF) Core Specification, http://www.omg.org/spec/MOF/.

[53] Potel, M.: MVP: Model-View-Presenter the Taligent programming model for C++ and Java, *Taligent Inc*, (1996), pp. 20.

[54] Saha, D. and Mukherjee, A.: Pervasive computing: a paradigm for the 21st century, *Computer*, Vol. 36, No. 3(2003), pp. 25–31.

[55] Salehie, M. and Tahvildari, L.: Self-adaptive software: Landscape and research challenges, *ACM transactions on autonomous and adaptive systems (TAAS)*, Vol. 4, No. 2(2009), pp. 14.

[56] Schmidt, D. C.: Model-driven engineering, *COMPUTER-IEEE COMPUTER SOCIETY-*, Vol. 39, No. 2(2006), pp. 25.

[57] Sharan, K.: Model-View-Controller Pattern, *Learn JavaFX 8*, Springer, 2015, pp. 419–434.

[58] Shaw, M. and Garlan, D.: *Software architecture: perspectives on an emerging discipline*, Prentice Hall Englewood Cliffs, 1996.

[59] Smith, J.: WPF Apps With The Model-View-ViewModel Design Pattern. Retrieved April 26, 2013, 2009.

[60] Sokolova, K., Lemercier, M., Garcia, L., and Saint Luc, L. C.: Towards High Quality Mobile Applications: Android Passive MVC Architecture, *International Journal On Advancesin Software*, Vol. 7, No. 2(2014), pp. 123–138.

[61] Streichert, T., Koch, D., Haubelt, C., and Teich, J.: Modeling and design of fault-tolerant and self-adaptive reconfigurable networked embedded systems, *EURASIP Journal on Embedded Systems*, Vol. 2006, No. 1(2006), pp. 9–9.

[62] Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T.: Overview of the IBM Java just-in-time compiler, *IBM systems Journal*, Vol. 39, No. 1(2000), pp. 175–193.

[63] Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L.: A component-and message-based architectural style for GUI software, *IEEE Transactions on Software Engineering*, Vol. 22, No. 6(1996), pp. 390–406.

[64] Tokuda, H., Nakajima, T., and Rao, P.: Real-Time Mach: Towards a Predictable Real-Time System, *USENIX Mach Symposium*, 1990, pp. 73–82.

[65] Weiser, M.: The Computer for the Twenty-First Century, *Scientific american*, Vol. 265, No. 3(1991), pp. 94–104.

[66] Willebeek-LeMair, M. H. and Reeves, A. P.: Strategies for dynamic load balancing on highly parallel computers, *IEEE Transactions on parallel and distributed systems*, Vol. 4, No. 9(1993), pp. 979–993.

[67] 江坂篤侍, 野呂昌満, 沢田篤史, 繁田雅信, 谷口弘一: コンテキストアウェアネスを考慮した組込みシステムのためのアスペクト指向アーキテクチャの適用と実現, ソフトウェア工学の基礎, Vol. 23(2016), pp. 175–180.

[68] 山本修一郎: 要求工学 (第 102 回) 参照モデルに対する保証ケース, ビジネスコミュニケーション, Vol. 50, No. 4(2013), pp. 66–70.

謝辞

本研究を遂行するにあたり，多くの方々から，多大なご指導，ご支援をいただきました．南山大学 理工学部ソフトウェア工学科 野呂昌満教授，青山幹雄教授，阿草清滋教授に深く感謝致します．野呂教授には，全てのことにおいて共通して本質を考えることが重要性であることを教えて頂きました．研究のご指導を通して，人間としてのありかたについてもご指導いただきました．なかなか改善されない筆者に対して，見放さず様々な表現方法でご指導頂き，本論文をまとめることができました．心から感謝致します．野呂教授のご尽力とお人柄により，すばらしい研究環境でした．筆者が精神的に挫けそうになった時も，心配頂き，暖かい言葉をかけて頂きました．時に厳しく，時に暖かい言葉のお陰で諦めず続けることができました．青山幹雄教授，阿草清滋教授には，研究を進めるにあたって，数々の適切な御助言を賜り，本研究を評価する上で重要な観点を得ることができ，本論文をまとめることができました．心から感謝致します．

普段から研究指導を頂いた南山大学 理工学部ソフトウェア工学科 沢田篤史教授に深く感謝致します．また，沢田教授の指導のもと，チューリッヒで発表をしたことは，研究を進める上でとても良い強い刺激を頂きました．筆者の研究環境のために多くのご支援を頂きました．厚く御礼申し上げます．

多くの教員の方々に研究だけでなく，精神的にもご支援頂きました．南山大学 理工学部ソフトウェア工学科 張漢明准教授にも，日頃からお気遣い頂き，励まして頂きました．自分が現場から逃げそうになった時，助けて頂けたお陰で今があります．心から感謝致します．

本研究は，様々な開発経験を基に進めることができました．多くの苦しい経験を共にした開発者の皆様に御礼を申し上げます．筆者のこれまでとこれからの研究成果から，少しでも現場が改善させたいと思います．

最後に，研究を一緒に続けてきた多くの先輩方，在学生の皆様，日頃から心の支えとなり励ましてくれた特に妹を始めとする家族の皆様，辛い時に奮い立たせてくれた yim の皆様とその楽曲に心から感謝します．

主論文

[1] 江坂篤侍, 野呂昌満, 沢田篤史: SOA に基づくシステムのアーキテクチャと仕様モデルの対応関係, ソフトウェア工学の基礎, Vol. 21, 2014, pp. 175-180.

[2] 江坂篤侍, 野呂昌満, 沢田篤史: インタラクティブソフトウェアの共通アーキテクチャの提案, ソフトウェアエンジニアリングシンポジウム (SES2015) 論文集, Vol. 2015, 2015, pp.137-144.

[3] 江坂篤侍, 野呂昌満, 沢田篤史, 繁田雅信, 谷口弘一: コンテキストアウェアネスを考慮した組込みシステムのためのアスペクト指向アーキテクチャの適用と実現, ソフトウェア工学の基礎, Vol. 23, 2016, pp.175-180.

[4] Esaka,A., Masami, N., Sawada, A.: Design of Common Software Architecture as Base for Application Generator and Meta-Generator for Interactive Systems, *Computer Software and Applications Conference (COMPSAC) 2017 IEEE 41st Annual*, IEEE Computer Society, Vol.2, 2017, pp.323-328.

[5] 江坂篤侍, 野呂昌満, 沢田篤史, 繁田雅信, 谷口弘一: コンテキストアウェアネスを考慮した組込みシステムのためのアスペクト指向アーキテクチャの設計, ソフトウェア工学の基礎, Vol. 24, 2017, pp.3-12.

[6] 江坂篤侍, 野呂昌満, 沢田篤史: インタラクティブシステムのための共通アーキテクチャの提案, ソフトウェア工学の基礎, Vol. 24, 2017, pp.129-134.

参考論文

[1] 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, 研究報告ソフトウェア工学 (SE),Vol. 2013-SE-179, No. 25, 2013, pp.1-6.

[2] 江坂篤侍, 野呂昌満, 沢田篤史: SOA 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, 研究報告ソフトウェア工学 (SE),Vol. 2014-SE-183, No. 13, 2014, pp.1-8.

[3] 江坂篤侍, 野呂昌満, 沢田篤史: インタラクティブソフトウェアの共通アーキテクチャの提案, 研究報告ソフトウェア工学 (SE),Vol. 2015-SE-187, No. 32, 2015, pp.1-8.

[4] 江坂篤侍, 野呂昌満, 沢田篤史, 繁田雅信, 谷口弘一: 組込みシステムへのコンテキスト指向プログラミング技術の適用, 研究報告ソフトウェア工学 (SE),Vol. 2016-SE-193, No. 11, 2016, pp.1-8.