動的適応可能なソフトウェア・モデルのための言語的アプローチ

A Linguistic Approach for Dynamically Adaptable Software Model

天野 憲樹
Noriki AMANO
n-amano@jaist.ac.jp

渡部 卓雄
Takuo WATANABE
takuo@jaist.ac.jp

北陸先端科学技術大学院大学 情報科学研究科*
Graduate School of Information Science
Japan Advanced Institute of Science and Technology

概要

開放型分散システム環境や移動計算機環境に柔軟に適応できるソフトウェア・モデルおよびそれを実現するためのプログラミング言語について述べる．

1 はじめに

本稿では，開放型分散システム環境および移動計算機環境に有効な動的適応能力を備えたソフトウェア・モデルとそれを実現するためのプログラミング言語を提案する．開放型分散システム環境や移動計算機環境の普及にともない，プログラマは設計の段階でそのソフトウェアがどのような実行環境上で使用されるか，あるいは実行時にその実行環境がどのように変化するかを完全に予測することができなくなった．こうした実行環境やその状態の変化がプログラムの実行に与える影響は無視できない場合が少なくない．例えば，*1)* Netscape ような比較的大きなアプリケーションの場合，デスクトップ環境では問題なく使用できるものの，計算機資源の極めて限定された携帯情報端末で使用することは難しい．*2)* 通信の形態を有線から無線に切替えた場合，一般に回線を長時間占有するタイプのアプリケーション (電子会議システムなど) はうまく動作しない．このような状況をうまく扱うにはアプリケーションの振舞いを実行環境やその状態に応じて変える必要がある．しかし個々の実行環境やその時々の状態に特化したアプリケーションを作成することは不可能である．こうした実行環境やその状態の変化は事前に予測できないため，ソフトウェア自身がそのような変化に動的 (実行時) に適応できる能力を持つことが望まれる．本稿では，このようなソフトウェアの能力を動的適応可能性と呼ぶ．動的適応可能なソフトウェアは，各計算機の持つ計算能力，資源などの特性を最大限に活かすために，それ自身の機能の柔軟な変更を可能とする．

プログラマが事前にソフトウェアの実行環境とその状態を特定することができない以上，事前にソフトウェアに持たせることができる動的適応可能性にも自ずと限界がある．このため動的適応可能なソフトウェアは拡張が容易かつ安全な機構を持ち，それを用いて利用者が適宜，その動的適応可能性を向上できるように構成される必要がある．本稿では，このような拡張の機構をも含めたより広い意味で動的適応可能性という語を用いる．

本研究の鍵となるアイディアは，*1)* 手続きにもとづく動的適応可能性 (手続き呼び出しにおいて，実行環境の状態に応じて呼ばれる手続きのコードを動的に切替える) を実現する，*2)* メタレベル・アーキテクチャを用いてソフトウェアを構成し，動的適応可能性のための記述を問題領域の記述から分離・独立させる，といった点にある．

本稿の構成は以下のとおりである．2 節で本研究の提案するソフトウェア・モデルの基本概念を概説す

*923-1292 石川県能美郡辰口町旭台 1-1

る．3 節ではこのソフトウェア・モデルを記述する言語の必要性について述べる．4 節で本研究の提案する言語 LEAD++について概説し，5 節で LEAD++によるプログラミングの記述例を示す．6 節では関連研究について述べる．そして 7 節で今後の予定について述べる．

2 基本概念

本研究で提案する動的適応可能なソフトウェア・モデルの鍵となるアイディアを以下に示す．

- 動的適応可能性を実現する機構は適応可能手続きに基づく．適応可能手続きが呼ばれると，この手続きの実際のコード（適応可能メソッド）が実行環境の状態に応じて動的に選択され実行される．この実行環境の状態に適した適応可能メソッドの選択方針を適応の戦略と呼ぶ．
- 適応可能手続きおよび適応可能メソッドを独立性の高いモジュールとして実現する．
- 動的適応可能なソフトウェアをメタレベル・アーキテクチャとして構成し，動的適応可能性のための記述（適応の記述）を問題領域の記述（問題の記述）から分離・独立させる．

つまり 1 つの適応可能手続きは実行環境の状態に適した複数の適応可能メソッドを持ち，適応可能手続き呼び出し時に，その時々の実行環境の状態に適した適応可能メソッドが選択され実行される．これらの適応の記述はメタレベルの記述として実現され，ベースレベルの記述（問題の記述）から分離・独立される．この方式には以下の利点がある．

1. 手続きに基づいて動的適応可能性を容易に実現できる．
2. 適応可能メソッドの容易かつ安全な修正・追加ができ，動的適応可能性を容易に向上できる．
3. 適応の記述の共有や再利用が容易になる．
4. 適応可能メソッドの内容（記述）を増減させることで，粗粒度から細粒度まで規模の柔軟な動的適応可能性を実現できる．
5. 既存言語[1]により記述されたソフトウェアへの適用が可能である．
6. 動的適応可能性の制御機構が単純になり，その制御にかかるコストを軽減できる．

[1] 手続き呼び出しの機能を持つ言語．

2.1 動的適応可能性の定義

本研究のソフトウェア・モデルの動的適応可能性は実行環境の状態，適応可能手続き，適応の戦略から実現される．本節ではこれらの定義を行なう．定義に先立ち，本稿では実行環境の状態とは以下の全てを含む広い意味で用いることとする．

- ハードウェアの構成要素の状態．
- ソフトウェアの構成要素の状態．
- 計算機内部のプロセスの状態．
- プログラムの入出力データ等の状態．
- 計算機周囲の物理的/論理的な環境状態．

定義 **1**（実行環境の状態）実行環境を構成する要素（環境要素）は有限個であり，ここではその数を N とする．そして $Value_i\ (i = 1, \cdots, N)$ を各要素のとり得る値（状態）の集合とすると，実行環境の状態 $Rstate$ は以下のように定義される．

$$Rstate = Value_1 \times \cdots \times Value_N$$

ただし，$1 \leq N$ ■

つまり $Rstate$ はある実行環境における遷移可能な状態の集合であり，その元はある時点の実行環境の状態を示す．

定義 **2**（適応可能メソッド）$StateCond$ を適応可能メソッドが選択される条件（ディスパッチ条件）の集合，$Behavior$ を振舞いの集合とすると，適応可能メソッドの集合 $Amethod$ は以下のように定義される．

$$Amethod = StateCond \times Behavior$$

ここで，$StateCond$ は，

$$StateCond = Cond_1 \times \cdots \times Cond_N$$

のように定義される．$Cond_i\ (i = 1, \cdots, N)$ の要素は i 番目の環境要素に関する条件であり，$Value_i$ から 0（偽）または 1（真）への関数として以下のように表現される．

$$Cond_i = Value_i \rightarrow \{0,\ 1\}$$

■

定義 **3**（適応可能手続き）$Amethod$ を適応可能メソッドの集合とすると，適応可能手続きの集合 $Aproc$ は以下のように定義される．

$$Aproc = Amehtod^{+}$$

ここで，$Amehtod^{+}$ は適応可能メソッド 1 個以上の順序の付いた並びであるとする． ■

定義 **4**（適応の戦略）$Rstate$ を実行環境の状態の集合，$Aproc$ を適応可能手続きの集合とすると，適応の戦略の集合 $Strategy$ は以下のように定義される．

$$Strategy = Rstate \times Aproc \rightarrow Amethod$$

■

つまり適応の戦略とは基本的には実行環境の状態と適応可能手続きを引数にとり，その状態に合致する条件を持つ適応可能メソッドを返す関数である．言うまでもなく適応の戦略は重要である．この実現方法によってソフトウェアの動的適応能力に差が生じるからである．ここで例として，以下のような戦略を定義する．

例：（各種戦略の定義）ある時点における実行環境の状態を σ とし，その時点において適応可能手続き ρ が起動されるとする．ここで，$\sigma \in Rstate$，$\rho \in Aproc$，$m_i \in Amethod$，$\gamma_i \in StateCond$，$v_j \in Value_j$，$c_j^i \in Cond_j$ のようであるとする（ただし，$1 \leq i \leq k,\ 1 \leq j \leq N$）．

$$\begin{aligned} \sigma &= (v_1, \cdots, v_N) \\ \rho &= (m_1, \cdots, m_k) \\ m_i &= (\gamma_i,\ \beta_i) \\ \gamma_i &= (c_1^i, \cdots, c_N^i) \end{aligned}$$

適応の戦略とは適応可能手続き ρ 中から実行環境に適した適応可能メソッドを 1 つ選択する方針である．ここでは例として，以下の戦略をあげる．

and 戦略：適応可能メソッドのディスパッチ条件が，

$$\forall j.c_j^i(v_j) = 1$$

となる（全て真となる）ものを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最も小さいものを選択する． ■

or 戦略：適応可能メソッドのディスパッチ条件が，

$$\exists j.c_j^i(v_j) = 1$$

となる（1 つ以上の条件が真となる）ものを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最も小さいものを選択する． ■

正の優先度数戦略：環境要素に重み w_j を付け，対応する条件が真となる環境要素の重みの総計

$$\sum_{j=1}^{n} w_j \cdot c_j^i(v_j)$$

が最大であるものを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最も小さいものを選択する． ■

負の優先度数戦略：環境要素に重み w_j を付け，対応する条件が偽となる環境要素の重みの総計

$$\sum_{j=1}^{n} w_j \cdot (1 - c_j^i(v_j))$$

が最小であるものを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最も小さいものを選択する． ■

言うまでもなく，ここにあげた戦略はほんの一例に過ぎない．より重要なことはさまざまな戦略をこのように定義できること，そして戦略の実現方法により異なる動的適応可能性を実現できる点にある．

2.2 適応の戦略と適応能力

本研究で提案する動的適応可能性の機構は単純であるが，適応の戦略の実現方法によって十分な適応能力を実現することができる．本節では適応の戦略とその適応能力について考察する．例として以下のような実行環境の状態 r，適応可能手続き p および適応可能メソッド（$m_1,\ m_2,\ m_3$）を仮定する．

$$\begin{aligned} r &= \{v_1,\ v_2,\ v_3\} \\ p &= \{m_1,\ m_2,\ m_3\} \\ m_1 &= (s_1,\ b_1) \quad (s_1 = (c_1^1,\ c_2^1,\ c_3^1)) \\ m_2 &= (s_2,\ b_2) \quad (s_2 = (c_1^2,\ c_2^2,\ c_3^2)) \\ m_3 &= (s_3,\ b_3) \quad (s_3 = (c_1^3,\ c_2^3,\ c_3^3)) \end{aligned}$$

実行環境の全ての状態を事前に知ることはできないので，予測可能な実行環境の状態は有限であり，

それに適した適応可能メソッドの定義も有限である．ここでは $m_1,\ m_2,\ m_3$ の 3 つのメソッドを定義している．これらの仮定の上で以下の 3 つの適応の戦略について考察する．

1. $\boldsymbol{strategy_1}$: 実行環境の要素に対する条件が全て真となる適応可能メソッドを選択する．
2. $\boldsymbol{strategy_2}$: 実行環境の要素に対する条件が全て真となる適応可能メソッドを選択するが，該当するものがない場合，デフォルト適応可能メソッドとして m_1 を選択する．
3. $\boldsymbol{strategy_3}$: 実行環境の要素に対する条件の一部が真となる適応可能メソッドを選択する．

ここで，以下のような 3 つの実行環境の状態について考察する．

	θ_1	θ_2	θ_3
状態 1	(1, 1, 0)	(0, 0, 0)	(0, 0, 0)
状態 2	(0, 0, 0)	(1, 1, 0)	(0, 0, 0)
状態 3	(0, 0, 0)	(0, 0, 0)	(1, 1, 0)

ここで，

$$\begin{aligned}\theta_1 &= (c_1^1(v_1),\ c_2^1(v_2),\ c_3^1(v_3))\\ \theta_2 &= (c_1^2(v_1),\ c_2^2(v_2),\ c_3^2(v_3))\\ \theta_3 &= (c_1^3(v_1),\ c_2^3(v_2),\ c_3^3(v_3))\end{aligned}$$

とする．まず，$strategy_1$ は上記のいずれの状態にも適応できない (適応可能メソッドを選択できない) ．次に $strategy_2$ は上記のいずれの状態でも必ずデフォルトの適応可能メソッド m_1 を選択してしまう．しかし $strategy_3$ は以下のような適応可能メソッドの選択を実現することができる．

$strategy_1$	: (状態 1, p) →
	: (状態 2, p) →
	: (状態 3, p) →
$strategy_2$	: (状態 1, p) → m_1
	: (状態 2, p) → m_1
	: (状態 3, p) → m_1
$strategy_3$	: (状態 1, p) → m_1
	: (状態 2, p) → m_2
	: (状態 3, p) → m_3

仮定から考慮すると，この $strategy_3$ は $strategy_{1,2}$ よりも望ましく，その適応能力が高いと言える．しかし適応可能メソッドの処理内容によっては，$strategy_{1,2}$ が望ましい場合もあり得る．つまり適応の戦略を変えることで柔軟な動的適応可能性を実現することが可能になる．本研究では，適応の戦略そのものも適応可能手続きとして実現される．それゆえ戦略自体を実行環境の状態に応じて動的に変更することができ，柔軟かつ高い動的適応可能性を実現できる．また，適応可能メソッドの振舞いとして適応可能手続きを記述することができるため，多重の適応可能手続きを実現することも可能である．

2.3 記述の分離

本研究で提案する動的適応可能なソフトウェアは適応の記述 (適応の戦略を含む適応可能手続き) と問題の記述 (適応可能手続き呼び出しを含む) から構成される．こうした動的適応可能なソフトウェアを単層構造で実現した場合 (図 1-(a)) ，問題の記述と適応の記述が混在することになる．これらの記述の混在は以下の問題を引き起こす．

- プログラムの可読性と拡張性を低下させ，その保守を困難にする．
- 適応の記述の共有，再利用を困難にする．

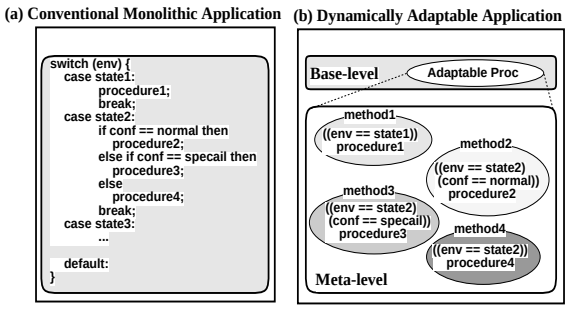

図 1: 記述の分離

このため本研究では，問題の記述をベースレベル，適応の記述をメタレベルとして，これらを分離・独立させる (図 1-(b)) ．このメタレベル・アーキテクチャにより，上記の問題が回避されるばかりでなく，適応の記述を容易かつ安全に拡張することが可能になる．そして適応の記述が完全に独立したモジュールである適応可能手続きおよび適応可能メソッドとして実現される点も，プログラムの可読性，拡張性，保守性の向上に貢献している．つまりベースレベルの記述や他の適応可能手続き，適応可能メソッドに何ら影響を与えることなく，適応可能手続きや適応可能メソッドの修正・追加が可能であり，これらの修正・追加により，適宜ソフトウェアの動的適応可能性を向上させることができる．

また適応の戦略が適応可能手続きとして実現されることも，動的適応可能性の向上に大きく貢献している．前述したように適応の戦略を適宜変えることで，ソフトウェアの適応能力を高めることができる．また適応の戦略の修正・追加を適応可能メソッドの修正・追加として容易に実現できる．適応可能手続きや適応可能メソッドは完全に独立したモジュールとして実現されるため，戦略の修正・追加は他の適応可能手続きや適応可能メソッドに影響を与えることなく安全に行なうことができる．戦略を変えることで，既存の適応可能手続きを修正することなく，その適応能力を柔軟に変えることが可能になる．以上の点から本研究で提案するソフトウェア・モデルの動的適応可能性は容易かつ安全に拡張が可能なかたちで実現されていると言える．

3 記述言語の必要性

本研究で提案する動的適応可能なソフトウェアを実現するためには，その記述言語が必要である．既存言語により本研究のソフトウェア・モデルを実現することも不可能ではないが，それを実現するために，かなりの注意と労力を要する．既存言語は，問題の記述と適応の記述をモジュール化する機構が欠如している．そして適応の記述を共有，再利用，拡張の容易な適応可能手続きおよび適応可能メソッドとして実現する機構も欠如している．同様に適応の戦略を適応可能手続きとして実現する枠組も欠けている．しかし，本研究のソフトウェア・モデルを実現するために，完全に新しい言語を用意しなければならない，というわけではない．既存の言語を拡張することで，これらを実現することも可能である．本研究のソフトウェア・モデルは特定の言語に依存したり，限定されるものではなく，より汎用性の高いものである．

これまでに我々は本研究のソフトウェア・モデルを実現するための言語 LEAD を設計し，そのプロトタイプおよび幾つかの動的適応可能なアプリケーションの作成を行なった．LEAD は本研究のソフトウェア・モデルに要求されるほとんどの機構を提供している．LEAD のプロトタイプは Scheme を拡張したインタプリタ型言語として実装されたが，他の既存言語との共生が可能であり，既存言語で作成されたソフトウェアに動的適応可能性を付加することができる．LEAD による動的適応可能なソフトウェアはメタレベル・アーキテクチャを形成するだけでなく自己反映性を持ち，より柔軟な動的適応可能性を実現することが可能となっている．また，LEAD の特徴的な機能として，実行環境の状態に応じたメソッドのリモート実行機能や，実行環境の状態変化を非同期のイベントとしてアプリケーションに通知する非同期イベント機能などがある．詳しくは参考文献 [1] を参照されたい．

4 オブジェクト指向言語 LEAD++

現在，我々はこれまでの研究により得られた知見をもとに LEAD の問題点を解決すべく LEAD に CLOS 的なオブジェクト指向概念 [2] を導入したオブジェクト指向言語 LEAD++の設計を行なっている．LEAD による動的適応可能なソフトウェアの最大の問題点は，その実行時システム上に (仮想的な) 実行環境が唯一固定的に存在するモデルである点である．このため，アプリケーション中の各処理に適した抽象度の動的適応可能性の導入および記述が不可能であった．しかし実用的な規模のアプリケーションでは，同一アプリケーション内でも各処理ごとにその抽象度が異なることはごく一般的なことであり，この問題は柔軟な動的適応可能性を実現する上で大きな制約となっていた．LEAD++では実行時システム上に仮想的な実行環境を表現する環境オブジェクトを任意の数だけ生成し，アプリケーション中の各処理に割り当てることができる．環境オブジェクトは実際の実行環境の状態変化を反映するオブジェクトであり，環境オブジェクトへの操作はそのクラスに関する総称関数として実現されるため，このオブジェクトの抽象度は各処理の抽象度に合わせて定義することができる．

本研究のソフトウェア・モデルは手続きを中心とした動的適応可能性をその基本概念としているため，オブジェクトにその操作としてのメソッドが付随するオブジェクト・モデルよりも，総称関数単位にメソッドが束ねられている CLOS のオブジェクト・モデルの方が親和性が高い．LEAD++ではクラス，多重メソッドを伴う総称関数，そして LEAD++を構成する諸言語要素 (クラス/総称関数/ メソッド /適応可能手続き /適応可能メソッド等) を LEAD++自身のオブジェクトとして表現するメタオブジェクトなどが導入される．さらに CLOS 同様，LEAD++のメタオブジェクトの外部仕様をメタオブジェクトプロトコル (MOP)[2] という形式で提供する．そし

て LEAD++による動的適応可能なソフトウェアは LEAD++の MOP を用いて実現される．LEAD++は CLOS から大きな影響を受けているが，ソフトウェアの動的適応可能性を実現する上で以下の特徴的な点を持つ．

1. 実行環境の状態を取得するライブラリ関数群を提供する．
2. 環境オブジェクトとそのクラスに関する総称関数群を提供する．
3. 適応可能手続きと適応可能メソッドを実現する機構を持つ．

4.1 ライブラリ関数群

LEAD++は実行環境を構成する最小単位の要素（環境要素）の状態を取得するライブラリ関数群を備える．この関数群は以下のように大別される．

- ハードウェア環境の状態を取得する関数群．
- ソフトウェア環境の状態を取得する関数群．
- 計算機内部のプロセスの状態を取得する関数群．
- 計算機周囲の物理的/論理的な環境状態を取得する関数群．

個々の関数はその名前に対応する環境要素を表し，その返り値は数値である．これらの関数群を用いることで，適応可能手続きを実現する上で必要となる実行環境の状態を取得することができる．またこれらの関数群を組み合わせることで抽象度の高い実行環境の状態を表現することが可能となる．表 1 は，ハードウェア環境の状態を取得するライブラリ関数群の一部である．

関数名（環境要素名）	返り値（状態）
network	ネットワークの接続種別
memory	使用可能なメモリ容量
display	ディスプレイ・サイズ
power	電源種別
battery	バッテリ残量

表 1: ハードウェア環境に関するライブラリ関数群の一部

4.2 環境オブジェクト

環境オブジェクトは各アプリケーションあるいはその中の個々の処理に適した抽象度の実行環境を表現するオブジェクトであり，そのクラスに関する総称関数群は環境オブジェクトの表現する実行環境への操作を提供する．またこの総称関数群は抽象度の高い環境要素を定義するものととらえることができる．先に述べた LEAD++のライブラリ関数群は抽象度の低い個々の環境要素の状態を取得するものであり，各アプリケーションあるいはその中の個々の処理の抽象度に合致するとは限らない．そのため個々の環境オブジェクトが表現する抽象度の高い実行環境に適した環境要素，すなわちその実行環境の状態を取得する関数を各環境オブジェクトのクラスに関する総称関数として実現する．この総称関数の実現には LEAD++の提供するライブラリ関数群を使用する．つまり 2.1 節の定義 1（実行環境の状態）における $Value_i$（各環境要素のとり得る状態の集合）を総称関数として実現する．このような環境オブジェクトとその操作（総称関数）は適応可能メソッド中のディスパッチ条件の記述に用いられる．環境オブジェクトは同一アプリケーション中で任意の数だけ生成できることから，各アプリケーションごと，あるいは各処理ごとの抽象度に適した実行環境を表現することができる．これより LEAD の問題点を解決することが可能となる．また，この抽象度の高い環境要素は各環境オブジェクトのクラスに関する総称関数として実現されため，その名前が衝突することもない．

環境オブジェクト（$Eobj$）は LEAD++のメタオブジェクトとして実現され，その構成要素は図 2 のようになっている．

$$
\begin{array}{lll}
\langle Eobj \rangle & ::= & (\langle Name \rangle \langle Supers \rangle \langle Strategy \rangle \langle S\text{-}list \rangle) \\
\langle Name \rangle & ::= & \langle \text{ クラス名 } \rangle \\
\langle Supers \rangle & ::= & (\{\langle \text{ 親クラス名 } \rangle\}^*) \\
\langle Strategy \rangle & ::= & \langle \text{ 適応の戦略名 } \rangle \\
\langle S\text{-}list \rangle & ::= & (\{\langle \text{ スロット指定子 } \rangle\}^*)
\end{array}
$$

図 2: 環境オブジェクトの構成要素

環境オブジェクトにおいて $Strategy$ は必須スロットである．つまり適応の戦略は環境オブジェクトごとに指定され，このオブジェクトを実行環境と見なす適応可能手続きの（適応可能）メソッド選択方針を決定する．このスロットの値を変えることで，このオブジェクトに関係する適応可能手続きの振舞いを一度に変えることができる．また環境オブジェクトのクラスに関する総称関数の定義については第 5 節において，その具体例を示す．

4.3 適応可能手続き & 適応可能メソッド

適応可能手続きと適応可能メソッドも LEAD++ ではメタオブジェクトとして表現される．これらは機能的，構造的に通常の総称関数とそのメソッドに類似しているが同一ではない．適応可能手続きは適応可能メソッドへのインタフェースであり，適応可能メソッドはそれ自身のディスパッチ条件を示す実行環境に対する条件と手続きコードもしくは手続き名を持つ．図 3，4 は，それぞれ適応可能手続き ($Aproc$) と適応可能メソッド ($Amethod$) の構成要素を示している．

$$
\begin{array}{lll}
\langle Aproc \rangle & ::= & (\langle Name \rangle \langle Parm \rangle \langle Methods \rangle) \\
\langle Name \rangle & ::= & \langle \text{ 適応可能手続き名 } \rangle \\
\langle Parm \rangle & ::= & (\{\langle \text{ パラメータ } \rangle\}^*) \\
\langle Methods \rangle & ::= & (\{\langle \text{ 適応可能メソッド } \rangle\}^*)
\end{array}
$$

図 3: 適応可能手続きの構成要素

$$
\begin{array}{lll}
\langle Amethod \rangle & ::= & (\langle Condition \rangle \langle Body \rangle) \\
\langle Condition \rangle & ::= & \langle \text{ ディスパッチ条件 } \rangle \\
\langle Body \rangle & ::= & \langle \text{ 手続きコード } \rangle | \langle \text{ 手続き名 } \rangle
\end{array}
$$

図 4: 適応可能メソッドの構成要素

適応可能メソッドの構成要素における $Condition$ は 2.1 節の定義 2 (適応可能メソッド) における $StateCond$ (メソッドのディスパッチ条件) を表す．同様に，上記の $Body$ は定義 2 の $Behavior$ (実際の振舞い) を表現している．ただし，適応可能メソッドの振舞いとしては，実際のコードを直接記述する方式以外に，実際のコードを実現している手続きの名前を記述することも可能とする．

4.4 適応の戦略

適応の戦略は各適応可能メソッド中のディスパッチ条件を評価し，適切なものを決定する．この評価の方法は各適応可能メソッドのディスパッチ条件で使用される環境要素の数，抽象度，環境要素間の関係などに依存する可能性があり，また最適な戦略が個々の適応可能手続きごとに異なる場合もあり得る．このため適応の戦略は適応可能手続きごとに動的に切替え可能であることが望まれる．しかし適応の戦略を共有できる適応可能手続きも少なくないと考えられる．このような理由から本研究では，適応の機構を 1 つの適応可能手続きとして実現し，各戦略は適応可能メソッドとして実現する．

前述したように，プログラマが事前に予測可能な実行環境とその状態は限定されるため，事前に定義できる適応可能メソッドも必然的に限定される．適応の戦略を適宜変更することで，限られた有限の適応可能メソッドをより有効に活用し，その動的適応可能性を向上させることができる (2.2 節参照) .

4.5 メタオブジェクトのクラス階層

LEAD++におけるメタオブジェクトのクラス階層は CLOS のそれに近い形で実現される (図 5) . 環境オブジェクトは `standard-env-object` クラスのインスタンスであり，適応可能手続きと適応可能メソッドは `standard-adaptive-procedure` クラス，`standard-adaptive-method` クラスのインスタンスである．`standard-env-object` クラスはハードウェア環境やソフトウェア環境などをモデル化するメタオブジェクトのクラスのサブクラスとして，また適応可能手続きと適応可能メソッドのクラスはそれぞれ，`standard-generic-function` (総称関数のクラス) , `standard-method` (メソッドのクラス) のサブクラスとして実現される．

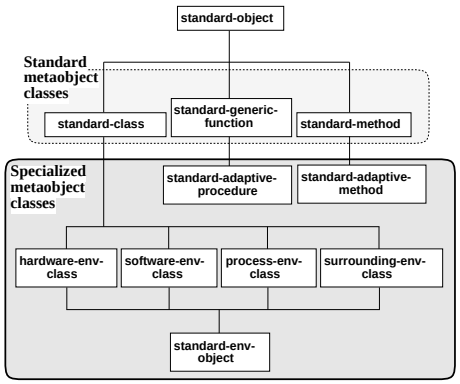

図 5: LEAD++におけるメタオブジェクトのクラス階層

4.6 LEAD++による動的適応可能なソフトウェアの構造と動作

LEAD++による動的適応可能なソフトウェアはメタレベル・アーキテクチャを形成する (図 6) . ベースレベルは対象ソフトウェアの問題の記述，環境オブジェクトの生成，適応可能手続き呼び出しからなり，メタレベルは環境オブジェクト (そのクラスに関する総称関数を含む) , 適応可能手続き (適応の

戦略を含む) の記述からなる .

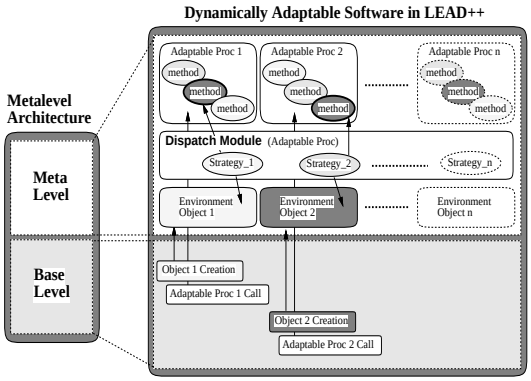

図 6: LEAD++による動的適応可能なソフトウェア

ソフトウェアの動的適応可能性を実現するために必要となる各レベルの記述は以下のようになる .

- メタレベル記述
  1. 環境オブジェクトとその操作の定義 .
  2. 適応可能手続きと適応可能メソッド定義 .
- ベースレベル記述
  1. 環境オブジェクトの生成 .
  2. 適応可能手続き呼び出し .

まずメタレベルでは環境オブジェクトのもとになるクラスの定義を行う .そしてそのクラスのオブジェクトの操作を総称関数として定義する . この操作とは具体的には環境オブジェクトの表現する仮想的な実行環境の状態を取得する関数である . 環境オブジェクトならびにその操作は , これを使用する各アプリケーションあるいはその中の各処理の抽象度に応じて定義する . 次に実行環境の状態に応じて変化させたい振舞いを適応可能手続き (適応可能メソッド) としてメタレベルに定義する . この適応可能手続きの記述には適応の戦略の記述も含まれる . しかし LEAD++ の実行時システムは幾つかの戦略を組み込みの戦略として提供する .

ベースレベルではメタレベルで定義された環境オブジェクトのインスタンス生成 , 適応可能手続きの呼び出しを行なう . 実行環境の状態に適した適応可能メソッドの選択は適応の戦略を実現するメタオブジェクト (戦略オブジェクト) により実現される . 環境オブジェクトには使用すべき適応の戦略が指定されている . まずこの指定により使用する戦略が決定される . 次にその戦略オブジェクトが (適応可能手続き中の) 適応可能メソッドの中に記述されたディスパッチ条件 (生成された環境オブジェクトの状態に対する条件) を評価し , 戦略に基づいて適切なものを選択する . そして選択された適応可能メソッドの $Body$ 部が評価され , 呼び出されたベースレベルのコンテキストで実行される .

メタレベルとベースレベルの区別は概念上のもので , 実現上は同一のアドレス空間に存在する . これによりユーザプログラム内からメタオブジェクトへのアクセスには特別な言語機構は必要なく , またカスタマイズしたいメタオブジェクトに関するクラス (総称関数 , メソッド) 定義も , 通常の定義と同様に行なうことができる . このためベースレベルのユーザプログラムから環境オブジェクト中に指定された適応の戦略を動的に変更することが可能である . この結果は適応可能メソッドの選択に反映する . すなわちメタレベルの変更がベースレベルのプログラムの振舞いに反映することになる . このように LEAD++による動的適応可能なソフトウェアは自己反映性 [3, 4] を持つ . この自己反映性を利用して , アプリケーションが持つ機能を実行環境の状態に応じて一時的に制限したり , またそれを解除するといった柔軟な適応動作を実現することが可能になる .

5 LEAD++による例題の記述

LEAD++による動的適応可能なソフトウェアのメタレベル記述は以下の手順で行う .

1. 環境オブジェクトのクラス定義 :
   `standard-env-object` のサブクラスを定義 .
2. 環境オブジェクトの操作を定義 :
   1 で定義したクラスに関し , LEAD++の MOP に従って総称関数とメソッドを定義 .
3. 適応可能手続きと適応可能メソッドの定義 :
   1 , 2 で定義したクラスと総称関数を使って適応可能手続きと適応可能メソッドを定義 .

5.1 動的適応可能なメーラ

LEAD++によるプログラミングの例として動的適応可能なメーラ (図 7) の記述をとりあげる . このメーラはネットワーク (環境要素) の接続状態 (有線接続/無線接続) に応じてその振舞いを変える . 有線接続時は通常の動作を行うが , バンド幅の狭い無線接続時には特定の人間からのメールだけを取得するように振舞いを変える .

下記のコードはこのメーラのメタレベル記述の一

部である．記述は CLOS 風の構文を用いている．ここで，defclass はクラスを，defmethod はメソッドを，def-adapt-proc と def-adapt-method はそれぞれ適応可能手続き，適応可能メソッドを定義するためのマクロとする．

```
;;; 動的適応可能なメーラのメタレベル記述 ( 一部 )
;;; 環境オブジェクトのクラス定義
(defclass new-env
          (standard-env-object) ())
;;; new-env の総称関数 ( 環境要素 ) 定義
(defmethod network-state ((obj new-env))
  (case (network) ;;; network はライブラリ関数
    (0 'wired)
    (1 'wired)
    (2 'wireless)
    (otherwise 'unknown)))
;;; 適応可能手続きの定義
(def-adapt-proc adapt-mailer (obj args ...))
;;; 適応可能メソッド 1 の定義 ( 有線接続時の振舞い )
(def-adapt-method adapt-mailer
  ((obj new-env) args ...)
  :condition
    ((eq (netowrk-state obj) 'wired)))
  :body
    ( ... )
      ...
    ( ... ))
;;; 適応可能メソッド 2 の定義 ( 無線接続時の振舞い )
(def-adapt-method adapt-mailer
  ((obj new-env) args ...)
  :condition
    ((eq (netowrk-state obj) 'wireless)))
  :body
    ( ... )
      ...
    ( ... ))
```

まず環境オブジェクトのもととなるクラス new-env を定義する．そしてこのクラスに関する操作としてネットワークの接続状態を取得する総称関数の定義を行なう．上記記述中の network は LEAD++の提供する低レベルのライブラリ関数である．この関数の抽象度は低いため，これをそのまま用いると記述の可読性を低下させる原因になる．これは比較的簡単な例であるため，それほど可読性に影響を与えないが，複数の環境要素からなる状態を表現する場合，このような低レベルの関数では記述が繁雑になることは明らかである．総称関数として抽象度の高い環境要素を定義することで，こうした問題が回避される．上記の記述中では適応可能メソッド 1 は有線接続時のメーラの振舞いを定義し，適応可能メソッド 2 は無線接続時のメーラの振舞いを定義している．

また下記の記述はこのメーラのベースレベル記述の一部である．ここで，make-instance は CLOS 同様，オブジェクトを生成する総称関数とする．また適応可能手続き呼び出し中の引数 args は適応可能メソッドの *Body* 部への引数であるとする．

```
;;; 動的適応可能なメーラのベースレベル記述 ( 一部 )
;;; インスタンス生成
(setq env-obj (make-instance 'new-env))
;;; 適応可能手続き呼び出し
(adapt-mailer env-obj args ...)
```

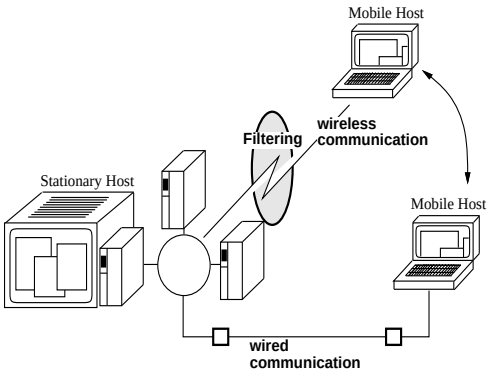

図 7: 動的適応可能なメーラ

6 関連研究

動的適応可能なソフトウェアを支援する言語的なアプローチとしてこれまでにも幾つかの研究がなされている．Rome[5] や Clovers[6] ではオブジェクトがクラスに基づく複数の表現を持つことができる．そしてオブジェクトにこれらの表現を動的に付加・削除することが可能である．しかしこれらの表現のもととなるクラスが既に定義されている必要があり，これらのクラスを動的に作ることはできない．適応化コンポジション [7] では，オブジェクトがある環境に入るとその環境に用意された振舞いを取得し，その環境に適したオブジェクトに変換される．しかし，適応化コンポジションでは，環境そのものが動的に変化することは考慮されていない．Gaea[8] はセルと呼ばれるプログラム片の構造がプロセス (実行時のプログラム) にとっての環境となり，その動的な変化にプロセスが適応できるが，環境そのものを明示的に表現することができない．AL-1/D[9] では，メタオブジェクトは環境要素のようなシステムの統計情報を持っている．そしてこの情報に対する制約を解消することで，メタオブジェクトのメソッドを自動的に実行することができる．しかしこの統計情報はシステムによって固定されたものであり，その数も少なく，その抽象度も低い．

LEAD++でも適応可能メソッド中にそのディスパッチ条件として実行環境の状態をある程度記述す

る必要がある．しかし，その記述が完全である必要はない．適応の戦略によりその記述の不完全さを補うことができるからである．上記の関連研究は本研究のように適応の戦略の部分を柔軟にカスタマイズできる機構が十分ではない．またそれらは，その記述言語に依存する度合が強く，それらのモデルを他の言語によるソフトウェアに導入することが難しい．本研究で提案するソフトウェア・モデルは単純で特定の言語に依存しないため，さまざま言語に適用することが可能である．

7 今後の予定

現在 LEAD++の詳細な言語仕様と実装方式について検討している．実装のプラットフォームとしては本研究のモデルに親和性の高い CLOS やこれまで開発してきた LEAD を予定している．また本稿で示した例題は非常に小規模かつ単純なものであるので，今後より規模の大きな動的適応可能なアプリケーションの記述を行ない，その評価を行なう必要がある．さらに移動コード型ソフトウェアへの動的適応可能性の導入も検討している．移動コード型のソフトウェアは実行環境の動的変化が著しい上に，その変化を予測することが非常に困難であるため，本研究で提案するソフトウェア・モデルの有効性を示すいい題材になり得ると考えている．

参考文献

[1] N. Amano and T. Watanabe: “Lead: Linguistic approach to dynamic adaptability for practical applications”, Proceedings of the System Implementation 2000, Chapman & Hall, pp. 277–290 (1998).

[2] G. Kiczales, J. des Rivieres and D. G. Bobrow: “The Art of the Metaobject Protocol”, The MIT Press (1991).

[3] P. Maes and D. Nardi Eds.: “Meta-level Architecture and Reflection”, ELSEVIER SCIENCE PUBLISHERS B.V. (1988).

[4] 渡部：“リフレクション”, コンピュータソフトウェア, **11**, 3, pp. 5–14 (1994).

[5] B. Carre, L. Dekker and J.-M. Geib: “Multiple and evolutive representation in the ROME language”, Proceedings of Technology of Object-Oriented Language and Systems (TOOLS’90) (1990).

[6] L. A. Stein and S. B. Zdonik: “Clovers: The dynamic behavior of types and instances”, Technical Report CS-89-42, Department of Computer Science, Brown University (1989).

[7] 本田, 渡, 所:“適応化コンポジション”, コンピュータソフトウェア, **9**, 2, pp. 48–62 (1992).

[8] 中島, 野田, 半田：“有機的プログラミング言語 gaea の設計と実装”, 日本ソフトウエア科学会第 13 回大会論文集日本ソフトウエア科学会, pp. 225–228 (1996).

[9] H. Okamura, Y. Ishikawa and M. Tokoro: “AL-1/d: A distributed programming system with multi-model reflection framework”, Proceedings of the IMSA’92 International Workshop on Reflection and Meta-level Architecture (1992).