モーバイルコンピューティングのための動的適応可能なソフトウエアアーキテクチャ

An Architecture of Dynamic Adaptable Software for Mobile Computing

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*
Department of Information Science
Japan Advanced Institute of Science and Technology

概要

移動計算機環境では計算機システムの状態がアプリケーションの実行中に著しく変化する可能性がある．このような移動計算機環境で使用されるアプリケーションは，その時々の状態の特性を最大限に利用できることが望ましい．本研究ではこうしたアプリケーションを実現するためのソフトウエアアーキテクチャを提案し，これを形成する言語 LEAD の実現を目指す．鍵となる考えは，計算機システムの状態に応じて実行時に振舞いを変えることができるアプリケーションを実現することである．このようなアプリケーションを (動的) 適応可能なアプリケーションと呼ぶ．LEAD は適応可能なアプリケーションの作成を支援するプログラミング言語であり，その実行時システムは計算機システムの状態変化を認識し，それにもとづいてアプリケーションの振舞いを変化させる機構を持つ．LEAD により作成された適応可能なアプリケーションはメタレベルアーキテクチャを形成する．これにより拡張性や再利用性の高い適応可能なアプリケーションの構築が可能となる．

1 はじめに

移動計算機環境ではアプリケーションの実行中に計算機システムの状態[1] (以下，システム状態) が変化する可能性がある．その要因として計算機移動 (host mobility) とコード移動 (code mobility) が考えられる．計算機移動とは，ユーザが計算機を持って移動することを指し，これにより計算機の物理的な位置やネットワークの接続形態が変化する．そして移動先の物理的な環境等に応じて PC Card などを適宜使い分けることが考えられるため，ハードウエア構成の変化も計算機移動により発生する変化ととらえることにする．またコード移動とは，他の計算機に動的に送られ実行されるプログラムコードの移動を指す．こうしたプログラムコードはモーバイルコードと呼ばれ，近年インターネットの普及にともないこれを支援する言語が多数開発されている．モーバイルコードによるアプリケーションは，実行される計算機の種類やシステム状態がその都度まったく違ったものになる可能性が高い．計算機の種類やシステム状態が著しく変化する場合，これらの変化がアプリケーションの実行に与える影響は無視できない．

このような移動計算機環境における変化に対して取り得るアプローチは 2 つの段階が考えられる．それらは (1) アプリケーションより下位のレベルで変化を吸収する，(2) 変化に従ってアプリケーション自身の振舞いを変える，である．これまでは主に (1) による対応が研究されてきた．しかし物理的に解決できない変化 (有線通信と無線通信におけるバンド幅など) もあり，完全に透過なシステム状態を実現することはできない．このように (1) で吸収できない変化に対して，アプリケーションが振舞いを変えることで得られる利点は少なくない．例えば，GUI ベースのアプリケーションはウインドウや文字のサイズ，ウィジェットの配置などを，使用するディスプレイサイズに応じて，適宜変えられることが望ましい．この振舞いの変化はウインドウや文字の見やすさと操作性の向上をもたらす．このようにアプリケーションは実行される計算機やシステム状態の特性を最大限に利用できるように実装されるべきである．しかし同一のアプリケーションを計算機の種類やシステム状態の数だけ個別に実装することは現実的ではない．

計算機の種類やシステム状態の変化に応じて，実行時の振舞いを変える能力 (機構) をアプリケーションに持たせることで，上記 (2) を効果的に実現することができる．この能力を (動的) 適応可能性と呼び，これにより変化する振舞いを適応可能な振舞い，さ

* 923-12 石川県能美郡辰口町旭台 1-1

[1] 本稿では計算機システムの状態とは，ハードウエア/ソフトウエアの構成とその状態，計算機をとりまく物理的な環境の状態など，広義の状態を指す．

らにこの適応可能な振舞いを持つアプリケーションを適応可能なアプリケーションと呼ぶ．適応の機構は各アプリケーションがアドホックに実装するのではなく，アプリケーションから分離して実現することが望ましい．この分離によりアプリケーションプログラム中で，本来目的とする処理と適応のための処理が混在しなくなる．この結果アプリケーションプログラムの可読性，保守性，そして拡張性の向上が期待できるようになる．LEAD は適応可能なアプリケーションを記述するための言語であり，LEAD の実行時システムは適応可能なアプリケーションを支援する有効な機能を持つ．LEAD の目標はシステム透過性を実現することではなく，各計算機，各システム状態の特性を活かすことができるアプリケーションを実現することである．

本論文の構成は次のとおりである．2 節で適応可能なアプリケーションの例とその実現に必要な要素について述べる．3 節では LEAD の概要について述べ，4 節で LEAD による適応可能なアプリケーションのアーキテクチャを示す．5 節で LEAD を用いた適応可能なアプリケーションの実現方法について述べ，6 節では LEAD による適応可能なアプリケーションの具体例を示し，それについて議論する．最後に 7 節で関連研究について述べる．

2 適応可能なアプリケーション

移動計算機環境で使用されるアプリケーションを適応可能なアプリケーションとして構成することで得られる利点は少なくない．例えば，以下のような例が考えられる．

- メーラ :ネットワークの接続が有線から無線に切替えられた時，メーラはメールをスプールに蓄積しておき，再度有線に切替えられた時，まとめてメールを送信する (転送効率の向上)．
- エディタ :フラッシュディスクカードが装着された時，エディタはそれを直ちに認識して，ハードディスクに代えてフラッシュディスクを使用する (バッテリの節減)．
- GUI ベースのアプリケーション :使用されるディスプレイサイズに応じてウインドウや文字のサイズ，ウィジェットの配置等を変更する (見やすさと操作性の向上)．

このような適応可能なアプリケーションを実現するために適応の機構は以下のものを含む必要がある．

1. システムの状態を認識する機構．
2. システム状態に応じたアプリケーションの振舞い．
3. アプリケーションと通信するためのインタフェース．
4. 適応のための戦略を実現する機構．
5. 適応のために必要とするシステム状態を新たに定義する機構．

以上から，以下の結論が得られる．上記 1, 4, 5 は複数のアプリケーションから共有できると考えられる．つまりこれらの機構を提供するサーバが必要である．そして上記 4, 5 を実現するための記述言語が必要となる．適応のための戦略とはどのようにアプリケーションの振舞いを変えるかを決定する機構であり，こうした機構は拡張可能な形で実現し，ユーザによるカスタマイズが可能であることが望まれる．この戦略のカスタマイズのために，戦略を記述する言語が必要である．またシステム状態を新たに定義できる必要もある．振舞いを変えるためにどのようなシステム状態を必要とするか，あるいはシステム状態をどのように解釈するかは各アプリケーションごとに異なる可能性が高い．例えば，バッテリ残量の変化に応じて振舞いを変える 2 つのアプリケーション A，B を考える．A はバッテリの残量があるか否か，という単位で振舞いを変えるかもしれないし，B は振舞いを変える単位としてバッテリ残量の百分率を必要とするかもしれない．このようにシステム状態は，それを使用するアプリケーションに合わせて定義 (記述) 可能である必要がある．抽象度が高く扱いやすいシステム状態を定義するためにもシステム状態の記述言語が望まれる．この適応の戦略およびシステム状態の記述を十分に行うため，より記述力の高い言語が提供されるべきである．

また上記 2 を満たすために，システム状態とそれに応じたアプリケーションの振舞いの組を記述しやすい機構を提供する必要がある．システム状態とそれに応じたアプリケーションの振舞いは通常 if 文などによる条件分岐によって制御される．しかしこうした条件分岐の制御記述は多様なシステム状態に適応しなければならないアプリケーションの記述には向いていない．条件分岐の記述が深くネストすればプログラムの可読性と保守性は低下する．また条件分岐方式では予期しなかった新しいシステムの状態に容易に対応することが困難である．

そして上記 2 のアプリケーションの振舞いも同類のアプリケーション間で共有できる可能性がある．これを実現するにはアプリケーションの振舞いが高い独立性を持つモジュールとして実現される必要がある．

さらに複数のアプリケーションを支援するためにアプリケーションの記述言語から独立している必要がある．アプリケーションから独立するにあたりアプリケーションと通信を行うためのインタフェースも必要となる．

3 言語 LEAD の概要

LEAD は前節で述べた適応可能なアプリケーションを実現するための言語であり，アプリケーションに適応の機構を付与するための言語である．

LEAD はシステムの状態を取得する関数を多数持つ．そして LEAD は既存の言語で記述されたアプリケーションの振舞い (関数) を扱うことができる．つ

まり LEAD のコード中に既存のプログラミング言語のコードを埋め込むことができる．このため既存のアプリケーションを適応可能なアプリケーションに進化（拡張）させることが可能である．これはソフトウエア資産の有効利用，生産性の向上といった観点から重要である．

LEAD はシステム状態とそれに応じたアプリケーションの振舞いの組を制御するための記述として総称関数とメソッド・ディスパッチング方式を採用している．アプリケーションの振舞いは総称関数とそれに付随するメソッドという形式で実装される．そしてその制御は条件分岐による記述ではなく，メソッドのディスパッチ条件として個々のメソッド内に埋め込む方式をとっている．この方式によりアプリケーション中の振舞いの独立性が高まるだけでなく，適応可能な振舞いを容易に追加，拡張することが可能になる．またアプリケーションの振舞いは総称関数としてアプリケーションから独立するため，再利用やアプリケーション間での共有が可能となる．

LEAD を用いて適応のための戦略を記述することができる．ユーザは多様なシステム状態に適応するために戦略をカスタマイズすることができる．戦略の記述だけでなく，LEAD はシステム状態を新たに定義することも可能である．これにより多様なシステム状態に適応可能となるだけでなく，より抽象度が高く扱いやすいシステム状態の記述が可能になる．これらの記述を制限しないだけ十分な記述力を LEAD は持つ．

その他の LEAD の持つ特徴的な機能としては，システム状態の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，アプリケーションとの通信を可能にするプロセス間通信機能，そしてメソッドの遠隔実行機能などがある．

4 LEAD による適応可能なアプリケーションのアーキテクチャ

LEAD による適応可能なアプリケーションは，その振舞いのモジュール性，再利用性，拡張性を高めることができるような機構として実現される．LEAD はアプリケーション中の関数の振舞いを変えることで適応可能なアプリケーションを実現する．LEAD の実行時システムとアプリケーションはサーバ/クライアントかつメタ/ベースの関係にあり，LEAD による適応可能なアプリケーションはメタレベルアーキテクチャ[4] を形成する (Fig.1)．アプリケーション中の振舞い（関数）でシステム状態の変化に応じて変化すべきものを総称関数及びその個別の実装であるメソッドとして LEAD の実行時システムが提供するメタレベルに LEAD を用いて記述する．つまり適応可能な振舞いはアプリケーション本体から完全に分離される．これによりアプリケーションプログラムの可読性や保守性が高まるとともに，適応可能な振舞いのモジュール化が促進され，それによる適応可能な振舞いの再利用性と拡張性の向上が実現される．またこれらの適応可能な振舞いはアプリケーションから完全に独立しているため，複数のアプリケーション間で共有が可能となり，そしてアプリケーション本体に影響を与えずにインクリメンタルな振舞いの追加が可能となる．メタレベルに記述される適応可能な振舞いはアプリケーションの一部分を含むため，その実行コード部分（アプリケーション上で実現される実際の振舞い）はアプリケーションが記述されている言語で記述する．

LEAD の実行時システムはアプリケーションと通信するためのインタフェースを持つ．アプリケーションはこのインタフェースを利用することで，メタレベル上の総称関数呼び出しを行うことができる．アプリケーションから総称関数呼び出しが実行されると，LEAD の実行時システムはオペレーティングシステム (OS) から現在のシステム状態を取得し，この状態に適したメソッドを選択する．そしてメソッドの実行が行われる．このメソッド選択の戦略が適応可能な振舞い（総称関数）の変化を決定する．LEAD の実行時システムはこの戦略を複数用意している．この戦略自身も LEAD による総称関数として実装されており，ユーザによる追加，拡張が容易に可能となっている．またアプリケーションから実行時にこの戦略変更の要求を出すことができる．戦略変更の結果はアプリケーションのメソッド選択に必然的に反映する．つまり LEAD を用いた適応可能なアプリケーションは自己反映的 [4, 5] な振舞いを実現することができる．自己反映的な振舞いにより，アプリケーションの機能の一時的な制限やその解除といった操作がアプリケーション中のコードを修正することなく容易に実現できる．

適応可能な振舞い（総称関数）はシステムの状態が変化した時にだけ実行されるものとそうでないものに分けられる．LEAD の実行時システムはこれらを支援するため，前者に対してイベント駆動形式，後者に対して関数呼び出し形式の実行形式をそれぞれ用意している．前者は LEAD の非同期イベント機能を使用する場合に用いる．非同期イベント機構を用いることでシステム状態の変化に従って時々刻々と変化する適応可能なアプリケーションの記述が容易になる．アプリケーション中でシステム状態の変化を常に監視する必要がなくなからである．

LEAD を使用することで LEAD の実行時システムが持つ有効な機能を利用することが可能となる．

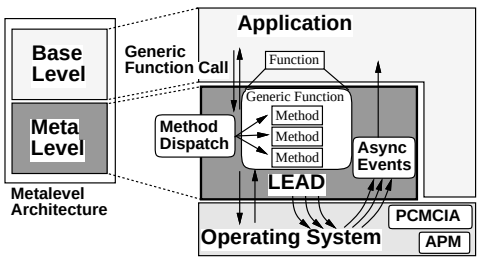

Figure 1: LEAD による適応可能なアプリケーションのアーキテクチャ

5 LEADによる適応可能なアプリケーションの実現

LEADによる適応可能なアプリケーションは，システム状態の定義，総称関数およびメソッドの定義，通信インタフェースによる総称関数の呼び出し，そしてシステム状態に応じたメソッドの選択により実現される．

5.1 システム状態の定義

LEADはアプリケーションが振舞いを変えるために要するシステム状態を定義することができる．LEADにおけるシステム状態とは，名前に関連付けられたシステムの状態を保持する属性[2]である．これには組み込みのシステム状態とユーザ定義のシステム状態がある．前者はLEADがあらかじめ提供している抽象度の低いシステム状態を指す．このシステム状態として，使用可能なメモリ容量を表すmemoryや，バッテリの残量を表すbatteryなどがある．後者は組み込みのシステム状態を組み合わせて定義する抽象度の高いシステム状態を指す．例えば，memoryとbatteryを組み合わせてemergencyというシステム状態を新たに定義することができる．ユーザによるシステム状態の定義を可能とすることで，システム状態の記述を簡略化し，システム状態を扱いやすくすることが可能となる．またこのシステム状態名は非同期イベント機能のイベント名としても使用される[3]．システム状態の定義はLEADの実行時システム上でLEADを用いて行う．

状態名	示す状態
network	ネットワークの接続
display	ディスプレイサイズ
power	電源種別

Table 1: 組み込みのシステム状態の例

5.2 総称関数とメソッド

総称関数とはシステム状態に応じて変化する振舞い(関数)の総称であり，個々の状態に応じた振舞いの実装であるメソッドへのインタフェースである．総称関数の定義はLEADの提供するdefgenericを用いて以下のように行う．

```
(defgeneric
   test ; 総称関数名
   emacs ; アプリケーション記述言語名
   (memory battery)) ; システム状態のリスト
```

この例はEmacs Lispによって記述されているtestという関数を総称関数として定義したものである．総称関数の定義にはその振舞いを決定するために必要とするシステム状態を指定する．この例では使用可能なメモリ容量を示すmemoryとバッテリ残量を示すbatteryをシステム状態として指定している．システム状態の指定は優先度の高いものを左から順に指定する．

メソッドは個々のシステム状態に応じた振舞いの実体であり、アプリケーションプログラムの一部を含む。通常1つの総称関数について複数の異なる振舞いのメソッドが存在する。メソッドの記述はシステム状態の記述 (システム状態に対する条件)とアプリケーションプログラムの振舞いが組になったものである．つまりLEADによる適応可能なアプリケーションの振舞いはこのシステム状態の記述にマッチしたメソッドを選択するメソッド・ディスパッチング方式である．

メソッドの定義はLEADの提供するdefmethodを用いて以下のように行う。

```
(defmethod
   test ; 総称関数名
   emacs ; アプリケーション記述言語名
   ((memory >= 10MB)(battery >= 50%))
   ; メソッドの選択条件リスト (状態の記述)
   local ; 実行ロケーション
   "(defun test (n) (* n n))") ; 実行コード
```

これは先の総称関数定義に対するメソッド定義の一例である．メソッドの選択条件リストとは，この総称関数に対して定義される複数のメソッドのうち，このメソッドが選択されるための条件 (システム状態)を記述したリストである．メソッドの選択は基本的にこのリストの条件を満たすか否かに応じてなされるが，このメソッド選択についてLEADの実行時システムはいくつかの戦略を用意しており，その戦略にもとづいてメソッドの選択を行う．実行ロケーションとは続く実行コードを実際に実行するホストを指定するものである．実行ロケーションにはlocalとremoteという2つの予約語がある．前者は現在ユーザが使用している手元のホストを指し，後者はデフォルトの遠隔ホストを指す．またここにホスト名を直接指定することもできる．選択されたメソッドの実行ロケーションがlocalである場合，そのメソッドは現在ユーザが使用中である手元のホストで実行され，remoteである場合はデフォルトの遠隔ホスト上で実行される．また直接ホスト名が指定されていた場合は，そのホスト上で実行される．このようなメソッドの遠隔実行機能は計算パワーの制限された携帯型計算機で負荷の大きなプログラムを実行する場合に有効である．遠隔地のより大きな処理能力を持つホストの計算パワーを得ることでアプリケーションの実行効率を高めることが期待できる．また，遠隔地のホスト上にあるデータベースの検索など，有効な応用が考えられる．これらの総称関数及びメソッドはアプリケーション本体から分離してLEADの実行時システムが提供するメタレベルに登

[2] 実際にはLEADにより記述された，値を1つ返す無引数の関数である．

[3] 現在の実装ではユーザが定義したシステム状態をイベントとして使用することはできない．

録する．実行コードとはアプリケーションに送られ実行される実際のプログラムコードである．そのためアプリケーションが記述されている言語で記述する．

5.3 通信インタフェース

LEAD の実行時システムはアプリケーションと通信を行うためのインタフェースを持つ．LEAD による適応可能なアプリケーションはこの通信インタフェースを使用してメタレベル上の総称関数呼び出しを行う．LEAD は LEAD 以外の言語で記述されたアプリケーションを適応可能なアプリケーションに拡張することができる[4]．このため通信インタフェースはアプリケーションが記述されているプログラミング言語に依存する．この通信インタフェースは抽象化され，アプリケーションからは通常の関数呼び出しと同等に実現されている．適応可能な振舞い（総称関数）の実行は関数呼び出し形式とイベント駆動形式に分けられ，それぞれの形式に応じたインタフェースが提供されている．関数呼び出し形式の実行はインタフェース関数`generic-func` を用いてアプリケーション中で以下のように記述する．

```
(generic-func '(総称関数名 引数 ...))
```

この実行によりアプリケーションから LEAD の実行時システムに対して総称関数の呼び出しがなされる．この動作には総称関数の要求，メソッドの選択，そして選択されたメソッドの実行までが含まれる．そのため総称関数に対する引数の指定も行う．この関数の返り値は指定した総称関数の返り値となる．

イベント駆動形式の実行は非同期イベントを使用するため 2 つのインタフェース関数を使用する．必要とするイベントの指定を`defevent` 関数で，イベントにより起動される総称関数を`eventfunc` 関数で指定する．

```
(defevent  '(イベント名 ...))
(eventfunc '(総称関数名 引数 ...) ...)
```

イベント，総称関数ともに複数の指定が可能であり，イベントと総称関数の対応付けをする必要はない．UNIX におけるシグナル関数と違い，LEAD の非同期イベント機構では 1 つのイベントに対して複数の総称関数を指定することができる．また複数のイベントに 1 つの総称関数を指定することもできる．イベントと総称関数の対応付けは実行時に動的になされるため，この指定方法はうまく機能することができる．この 2 つのインタフェース関数の実行により，アプリケーションでは指定したイベントの通知を LEAD の実行時システムから受けることができる．そしてイベントの通知を受けた時，指定した総称関数の呼び出しが自動的になされるが，LEAD のメソッド選択機構により現在のシステム状態に適合しないメソッドは選択されない．このイベント駆動形式の実行でもイベントの取得，総称関数の要求，メソッドの選択，そして選択されたメソッドの実行までが行われる．

その他のインタフェース関数としてはメソッド選択の戦略を指定する`generic-policy` 関数がある．

5.4 メソッド選択の戦略

システム状態にもとづくメソッドの選択機構は適応可能な振舞いのもっとも本質的な機構である．このメソッド選択の方針を戦略と呼ぶ．通常 1 つの総称関数に複数のメソッドが付随し，各メソッドごとにそのメソッドが選択される条件を記述するが，それらの条件をどのように解釈するかは戦略に依存する．個々のメソッドは他のメソッドの選択される条件については何も関知していない．つまり個々のメソッドの条件だけではメソッド全体の中から 1 つのメソッドを選択することができない．あるシステム状態では各メソッドの条件を満たすメソッドが複数存在するかもしれない．このような複数のメソッドの中から適切なメソッドを選択するアルゴリズムは，着目する観点によって大きく異なる．これは総称関数の性質にも関わる問題であり，かつシステム状態として何を想定し，それら状態間の関係（例えば，ハードウエア資源の優先順位付けなど）をどのように設定するかにも関わってくる．そのためこのメソッド選択の機構は拡張可能な形で実現されていることが望ましい．そうすることで総称関数の性質やシステム状態に対して，より適切なメソッドの選択（適応可能な振舞い）を実現する可能性を残すことができる．また戦略の変更により，既存の総称関数とそのメソッド群を修正することなく，新たな振舞いを実現することが可能になる．これは適応可能な振舞いの再利用性を高め，アプリケーションの開発コストの低減と生産性の向上につながる．LEAD の実行時システムは戦略を複数提供するとともに，戦略の拡張が可能な機構を提供している．つまりこの戦略自体が LEAD の総称関数として実現されているのである．このため既存の戦略の修正や新しい戦略の追加が容易に可能となる．LEAD がデフォルトで提供するメソッド選択の戦略には，単純戦略と優先度数戦略がある．これらを適宜使い分けることで，性質の異なる総称関数の実現が可能となる．

5.4.1 単純戦略

単純戦略は **AND** 戦略と **OR** 戦略に分けられる．前者は各メソッド定義中の選択条件リストを **AND** 条件として，後者はそれを **OR** 条件として解釈し，メソッドの選択を行う．つまり AND 戦略ではメソッドの選択条件リストの全条件が真となるメソッドを，OR 戦略では全条件のうち 1 つ以上の条件が真となるメソッドをそれぞれ選択する．これらの単純戦略は，ある特定のシステム状態でのみ実行する必要がある総称関数の実現に用いる．これらの総称関数は常に実行されるとは限らない．

[4] 現在の言語サポートは Emacs Lisp, Perl, Tcl/Tk, Python の 4 言語である．

5.4.2 優先度数戦略

優先度数戦略は総称関数定義時のシステム状態間に優先順による重み (優先度数) を付け，それらに対する各メソッドの選択条件の真偽にもとづく合計値によってメソッドを決定する．この優先度数戦略は，どんなシステム状態でも実行する必要のある総称関数の実現に用いる．つまりこれらの総称関数は必ず実行される．LEAD の実行時システムは総称関数の定義から 1 つの総称関数に対して 1 つの優先度数リストを生成する．優先度数リストはシステム状態名と，そのシステム状態に対する優先度数の対からなるリストである．各状態に対する優先度数は次の手順で算出される．ある総称関数 f で定義されるシステム状態の数を n とし，あるシステム状態の指定位置を k とすると，f におけるそのシステム状態に対する優先度数は 2^{n-k} となる．例えば，以下のような総称関数を定義した場合，

```
(defgeneric test
            emacs
            (network memory battery))
```

この総称関数に対して，以下の優先度数リストが生成される．

```
((network . 4)(memory . 2)(battery . 1))
```

そして優先度数戦略は以下の 2 つの規則にもとづいてメソッドを選択する．

A : 各システム状態に対する条件が偽となるものの優先度数の合計値 (負の合計) が最小であるものを優先する．

B : 各システム状態に対する条件が真となるものの優先度数の合計値 (正の合計) が最大であるものを優先する．

つまり規則 A により算出された合計値が同一であるメソッドが複数存在した場合、さらに規則 B を適用することでメソッドを決定することが可能になる．このようにメソッドを確実に決定するため，優先度数は 2 のべき乗の重み付けがなされている．アプリケーションプログラマは総称関数の定義時に優先すべきシステム状態の順に，システム状態の指定を行う．もしプログラマが優先しているシステム状態に対する条件が偽であるなら，そのメソッドを選択すべきではない．また最も優先しているシステム状態が真であるメソッドは，それ以外の優先度が低いシステム状態が真であるメソッドよりも優先されるべきである．ここで優先度数戦略にもとづくメソッド選択の例を示す．上にあげた総称関数定義に対して以下のメソッド及び計算機システムの状態を仮定する．

- モデムによるネットワークの接続．
- 使用可能なメモリ容量は 10MB．
- バッテリ駆動で，その残量は 50% ．

```
(defmethod example ; メソッド (1)
           emacs
           ((network = modem)
            (battery >= 50%)) ...)
(defmethod example ; メソッド (2)
           emacs
           ((network = wireless)
            (memory >= 10MB)
            (battery >= 50%)) ...)
```

上記の仮定より，メソッド (1) の選択条件の評価は以下のようになる．

```
(network  = modem) ; 真 (4) 負の合計 = 0
(battery >= 50%)   ; 真 (1) 正の合計 = 5
```

そしてメソッド (2) の選択条件の評価は以下のようになる．

```
(network  = wireless) ; 偽 (4) 負の合計 = 4
(memory  >= 10MB)     ; 真 (2) 正の合計 = 3
(battery >= 50%)      ; 真 (1)
```

上記の評価結果に規則 A を適用すると，この例ではメソッド (1) が選択される．

6 例題と議論

本研究では例題として Tcl/Tk を用いてテキスト・エディタとリソース・モニタを作成し，LEAD によりこれらを適応可能なアプリケーションに拡張した (Fig.2, 3) .この 2 つの例題は PC Card スロットを持ち，バッテリで動作するノート型パソコン上での使用を想定して作成した．LEAD のプロトタイプは Scheme に似た構文を持つインタプリタ型言語として実装した．また Emacs Lisp，Perl，Tcl/Tk，Python を用いて作成されたアプリケーションと LEAD の実行時システムが通信を行うためのインタフェースも実装した．これらの実装はすべて BSD/OS(Wildboar)[5] と SunOS[6] 上で行った．

6.1 適応可能なテキスト・エディタ

このエディタはシステム状態の変化に応じて以下のように振舞いを変える．

- 使用するディスプレイサイズに応じてメインウインドウ，ダイアログそして文字のサイズを実行時に変更する．さらにウィジェットのサイズと配置も変更する (見やすさと操作性の向上)．
- 使用可能なメモリ容量に応じて文字入出力バッファの伸縮を行う (メモリ容量の節減)．

[5] APM(Advanced Power Management) や PC Card サービスを支援している版．

[6] APM や PC Card サービスを支援していないため，ハードウエアの構成要素を示すシステム状態が制限される．

- バッテリ残量に応じてファイルの自動保存の回数を増減させる（ディスクアクセスの回数を制御し，バッテリの消費を節減）．この振舞いは時々刻々と変化するバッテリの残量を監視するためにLEADの提供する非同期イベントを使用している．

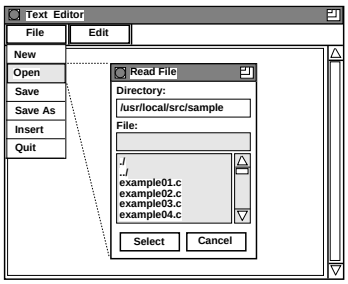

Figure 2: 適応可能なテキスト・エディタ

以下はファイル選択のダイアログを実現する記述の一部（LEADの実行時システム中に登録される総称関数とメソッドの記述）である．

```
(defgeneric ; 総称関数定義
   FileSelectDialog ; 総称関数名
   tcl ; アプリケーション記述言語名
   (display)) ; ディスプレイの状態
(defmethod ; メソッド (1)
   FileSelectDialog ; 総称関数名
   tcl ; アプリケーション記述言語名
   ((display >= LARGE))
    ; メソッドの選択条件リスト
   local ; 実行ロケーション
   "proc FileSelectDialog {w} {
         ...
    }" ) ; 実行コード (Tcl/Tk)
(defmethod ; メソッド (2)
   FileSelectDialog
   tcl
   ((display = SMALL))
   local
   "proc FileSelectDialog {w} {
         ...
    }" )
```

図中のメソッド(1)は大きいディスプレイの使用時に、メソッド(2)は小さいディスプレイの使用時に，それぞれその大きさに見あったサイズのダイアログとウィジェットおよびその配置を実現する．この総称関数の実行はアプリケーション中で以下のように行う。

```
generic_func "FileSelectDialog $w"
```

6.2 適応可能なリソース・モニタ

このモニタは現在のシステム資源の状態を表示する．そして電源の種別がバッテリであった場合は時々刻々と変化するバッテリの状態を監視するが，固定電源であった場合，バッテリの残量はあまり重要ではないので，それを逐一表示しないように振舞いを変える．このモニタも非同期イベントを使用して実現されている．

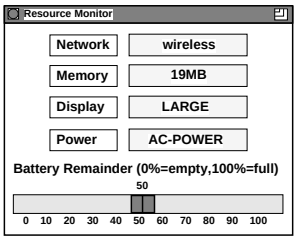

Figure 3: 適応可能なリソース・モニタ

以下はバッテリ残量を表示する振舞いの記述の一部（LEADの実行時システム中に登録される総称関数とメソッドの記述）である．

```
(defgeneric ; 総称関数定義
   BatteryDisp ; 総称関数名
   tcl ; アプリケーション記述言語名
   (power)) ; 電源の状態
(defmethod ; メソッド (1)
   BatteryDisp ; 総称関数名
   tcl ; アプリケーション記述言語名
   ((power = AC-POWER))
    ; メソッドの選択条件リスト
   local ; 実行ロケーション
   "proc BatteryDisp {w} {
         ...
    }" ) ; 実行コード (Tcl/Tk)
(defmethod ; メソッド (2)
   BatteryDisp
   tcl
   ((power = BATTERY))
   local
   "proc BatteryDisp {w} {
         ...
    }" )
```

図中のメソッド(1)は固定電源使用時に、メソッド(2)はバッテリ使用時に，それぞれ実行される．前述したように固定電源使用時はバッテリの残量を逐一表示しない．この総称関数の実行はアプリケーション中で以下のように行う。

```
defevent  "power"
eventfunc "BatteryDisp $win"
```

この記述により，使用する電源の種別が変わった時，その変化をLEADの実行時システムから非同期

のイベントとして受け取ることができる．そしてイベントの取得後，指定した総称関数 (BatteryDisp) が実行される．

6.3 議論

適応可能なアプリケーションの作成にかかる労力は LEAD を用いることによって軽減される．それは LEAD の実行時システムがシステム状態を認識する機構や適応可能な振舞いを制御するための戦略，非同期イベント機能などを持ち，LEAD を用いることでそれらを容易に利用することができるからである．例えば先の例題を LEAD を用いず Tcl/Tk だけで実現した場合を考慮すれば，LEAD の有効性は明らかである．Tcl/Tk だけで実現する場合，アプリケーション中でシステム状態を常に監視しなければならない．これは過度に CPU 時間を無駄にすることになる．またさまざまなシステム状態を区分けするために非常に深い条件分岐のネスティングを行う必要がある．これはプログラムの可読性と保守性を低下させる要因となる．これらから拡張性が低下することも明らかである．さらにメソッド選択の戦略を複数用意し，それらを適宜切替えるような仕様をアプリケーション中で実現するならば，アプリケーションの複雑さは一気に増大する．このようなアドホックな実装ではプログラムの再利用や生産性の向上は望めない．

LEAD を用いることで，上記の問題を解決することができる．先の例題では LEAD の非同期イベントを適宜用いることで，アプリケーションが常にシステム状態を監視する必要をなくしている．また適応可能な振舞い (総称関数) は完全にアプリケーション本体から分離されるので，アプリケーション中では本来目的とする処理の記述に専念することができ，プログラムは見通しのよいものになった．これにより保守性の向上が期待できる．このメタレベルアーキテクチャにより適応可能な振舞いをアプリケーションから分離することにはさらに利点がある．まず拡張性の向上があげられる．例えば，例題のエディタの場合，新たな大きさのディスプレイに見合うダイアログを実現する時，これまでのメソッドやアプリケーション本体に何ら影響を与えず，新しいメソッドをインクリメンタルに追加することで対応できる．このような高い拡張性は多様なシステム状態に柔軟に適応するアプリケーションには不可欠の属性である．そしてこの分離により総称関数の再利用やアプリケーション間での共有が可能となる[7]．例題ではディスプレイの大きさに応じたメインウインドウと文字のサイズを実現する総称関数をエディタとモニタの両方で共有させている．このような総称関数の再利用や共有によりアプリケーションの開発効率，生産性の向上が可能となる．

さらに例題のように LEAD を既存のアプリケーションの拡張用言語として用いる場合，LEAD を習得するために要するコストは比較的小さい．先の例題のコーディングを見れば明らかなように，既存のアプリケーションを拡張する場合，LEAD で記述するのはシステム状態とそれに応じた総称関数とメソッドだけであり，その記述の大半はアプリケーションの記述言語で記述できる．

[7] 現在の実装では，総称関数の共有は同一の言語で記述されたアプリケーション間のみに制限される．

7 関連研究

アプリケーションが実行される計算機の違いや，アプリケーション実行時のシステム状態の変化に対し，これまではオペレーティングシステム (OS) による対応が研究されてきた．これらの OS の目標はこのような変化を認識して，それらを吸収し，アプリケーションにシステムの透過性を提供することである．このような OS を適応可能な OS という．適応可能な OS として SPIN[1]，Exokernel[2]，Apertos[7] などがある．こうした適応可能な OS と LEAD は相補的な関係と位置付けられる．LEAD の目標はシステム透過性の実現ではなく，各計算機やシステム状態の特性を活かすことができるアプリケーションの実現である．このような LEAD を実現するには計算機の違いやシステム状態を認識する必要があり，OS レベルでこれらの認識を可能とする機構がサポートされている必要がある．しかし計算機を使用する周囲の環境の違いにより発生する物理的な差 (ネットワークのバンド幅など) に対して完全なシステムの透過性を実現することは不可能である．例えば，有線ネットワークと無線ネットワークの違いに対しては，OS により接続しているネットワークをアプリケーションに対して透過に見せることは重要であるが，それぞれのネットワークの特性に合った振舞いを実現することも重要である．クライアント/サーバ型のアプリケーションを有線ネットワークと無線ネットワークで使用する場合，前者は常にコネクションを張っても問題ないが，後者では狭いバンド幅や一時的切断の多発が考えられるため，常にコネクションを張る方式は問題がある．こうした場合，無線ネットワーク使用時には処理をまとめてサーバ側に送り，サーバ上で実行し，結果を受け取る方式に振舞いを変えることが考えられる．この方式は常にコネクションを張る必要がないので無線ネットワークに有効である．しかし常にこうしたリモート実行形式が優れているとは言えない．全ての場合にリモート実行形式を採用した場合，サーバ側のマシンに負荷が集中することになるからである．またこれらの OS が提供するシステム状態の認識機構は通常システムコールの形式で，抽象度の低いものである．アプリケーションが必要とする状態はより抽象度の高いものであることが多い．LEAD は抽象度の高いシステム状態の定義を可能とする言語である．

インターネットの普及にともないプログラムコードを他の計算機に送って実行させるモーバイルコードをサポートする言語が多数開発されている．実行コードを能動的に他の計算機に送り，そこで実行させる形式のモーバイルコードをサポートする言語として，Telescript[6] や Agent Tcl などがある．またサーバから実行コードをダウンロードして実行する種類のモーバイルコードをサポートする言語では Java[3]

や，Caml などがある．これらの言語は実行エンジン (実行時システム) を持ち，その実行エンジン上でそれらのアプリケーションを実行する．こうした実行エンジンはアプリケーションレベルで実現されるため，計算機や OS の種別によらないアプリケーションの実行を可能とする．しかしこうしたモーバイルコードによるアプリケーションは実行される計算機の種別やシステム状態を考慮して作成されないために問題が起こる場合がある．例えば，実行に大きなメモリ容量を必要とするアプリケーションをモーバイルコードで作成し，他の計算機上で実行することを考える．このアプリケーションを送った先の計算機のメモリ容量が少ない場合，このアプリケーションを実行することができなくなる．こうした状況は今後増加していくと考えられる．モーバイルコードによるアプリケーションを LEAD で包み込むことで，このような状況を回避することが可能となる．拡張用言語としての LEAD は対象とするアプリケーションが記述されている言語を限定しないので，LEAD によるモーバイルコード・アプリケーションの支援は可能である[8]．Telescript ではエージェントに適切なパーミットを与えることで，エージェントの行動を制限することができる．パーミットでは計算機資源や通信資源の時間的コスト，空間的コストなどの割り当てを行い，エージェントが無駄に資源を浪費するのを防ぐ．しかしこのパーミットはプログラム作成時に指定する必要があり，またパーミットの概念自体がセキュリティ等にまつわるエージェントの行動制限を目的とするものであるため，システム状態に応じて動的に振舞いを変えるエージェントの作成は困難である．

8 おわりに

LEAD を用いて実現された適応可能なアプリケーションはメタレベルアーキテクチャを形成し，適応可能な振舞いの再利用，拡張，共有が容易となる．これにより適応可能なアプリケーションの開発コスト削減と生産性の向上が可能となる．また LEAD を用いることによって，LEAD の実行時システムが提供するメソッド選択の機構や非同期イベント機構を既存のアプリケーションからでも利用することが可能になる．これは LEAD により既存のアプリケーションを適応可能なアプリケーションに拡張可能であるということを意味し，既存のソフトウエア資産の有効な再利用という観点から重要である．LEAD は LEAD 以外の言語で記述されたプログラムコードを LEAD のコード中に埋め込むことができるため，適応可能な振舞いの記述の大半はアプリケーションが記述されている言語で記述することができる．これは通常メソッドの定義において実際に振舞いを実現する実行コード部分の記述が大半を占めるからである．

さらに個々の適応可能な振舞い (総称関数) はアプリケーション本体から完全に分離されるので，アプリケーションの記述では，本来目的とする処理の記述に専念することができる．これはアプリケーション中に本来目的とする処理の記述と適応可能な振舞いを制御するための記述が混在することを回避し，アプリケーションプログラムの可読性と保守性を向上させる．

本稿では詳述することができなかったが，LEAD を用いてアプリケーションに自己反映的な機能を持たせることが可能である．これはメソッド選択の戦略を実行時にアプリケーションからの要求により動的に変更することで，アプリケーションの機能の一時的な制限やその解除を可能にする．

[8] 現在の LEAD の実装では Java のような中間コードにコンパイルする形式の言語を扱うには工夫する必要がある．

謝辞

本研究を行うにあたって貴重なご助言を頂いた北陸先端科学技術大学院大学の中島達夫、緒方和博の両先生ならびに同大学院言語設計学講座の諸氏に深謝する。本研究の一部は情報処理振興事業協会 (IPA) 「独創的情報処理技術育成事業」の一環として行われたものである。

References

[1] Brain N Bershad, Stefan Savage, Przemyslaw Pardyak, and Emin Gun Sirer. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, 1995.

[2] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1995.

[3] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-wesley publishing company, 1996.

[4] Pattie Maes. Computational reflection. Ph. D. Thesis 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[5] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 23–35, 1984.

[6] James E White. Telescript technology – mobile agents, 1991. February 1996.

[7] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pages 414–434, 1992.