LEAD：動的適応可能なソフトウェアを構成する言語の設計と実装

天野憲樹† 渡部卓雄†

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，さまざまな実行環境やその動的な状態変化に適応できるソフトウェアの必要性が高まっている．本稿では，そうした動的適応可能なソフトウェアのモデルおよびそれを構成するための言語LEADについて述べる．鍵となるアイディアは，*1*) 手続き呼び出し時に，手続きの実行コードを実行環境の状態に応じて動的に切り替える機構を導入する，*2*) 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，適応動作の記述と問題領域の記述を分離・独立させる，といった点にある．言語LEADは，このような動的に変化する手続きの定義，および制御の機構を言語機能として提供し，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．LEADを用いることで，*1*) 拡張性の高い動的適応可能なソフトウェアの実現，*2*) 既存ソフトウェアへの動的適応能力の導入，などが可能となる．

LEAD: The Design and Implementation of a Language for Constructing Dynamically Adaptable Software

NORIKI AMANO† and TAKUO WATANABE†

A system has *dynamic adaptability* if it can adapt itself to dynamically changing runtime environments. As *open-ended distributed systems* and *mobile computing systems* have spread widely, the need for software with dynamic adaptability increases. We propose a model of software with dynamic adaptability and, we designed and implemented the language LEAD which constructs the model. The basic idea that we introduce the mechanism which changes *procedure invocations* dynamically depending on the states of runtime environments. The dynamically adaptable software based on the model is realized as a *meta-level architecture*, and it separets the codes of dynamic adaptability from the codes of primary subject domain in the software. LEAD provides the mechanism as a language construct and, the dynamically adaptable software in LEAD forms meta-level architecture. Using LEAD, we can realize the followings: *1*) the highly extensible dynamically adaptable software, and *2*) the introduction of the dynamic adaptability to the existing software.

1. はじめに

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計算機上で使用されるか，あるいは実行時にその計算機環境がどのように変化するか（たとえば，計算機の移動やPCカードの着脱など）を完全に予測することが困難になった．またJava[5]やTelescript[12]などによるモーバイルエージェントは実行中にネットワークを通じて移動するため，移動前後でハードウェア環境，ソフトウェア環境，計算機周囲の物理的環境など（本稿では，これらすべてを含む広い意味で**実行環境**という語を用いる），すべてが変化する可能性がある．多くの場合，こうした実行環境の動的な状態変化がプログラムの実行に与える影響は無視できない．

さまざまな実行環境やその動的な状態変化に効果的に対処する方法として，ソフトウェアの振舞いを実行環境の状態に応じて動的に変えることが考えられる．たとえば，*1*) Netscape Communicatorのような比較的大きなアプリケーションの場合，一般に計算能力や計算機資源のきわめて限定された携帯情報端末（PDA）で使用することは難しい．しかし，あまり必要でない機能を一時的に制限する，画像の質を落とす，画像を表示しない等の処理を行うことでPDAでも使用できる可能性が高まる．またPDAのディスプレイは通常小さいため，デスクトップ環境のグラフィカル

† 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, Hokuriku

ユーザインタフェース（GUI）をそのまま用いると見にくく，操作性が悪くなる．これには文字やウインドウのサイズ，ウィジェット（部品）のサイズや配置等を変えることで対処できる．また，2) 通信の接続方式を有線から無線に切り替えた場合，リモートプロシジャコール（RPC）を用いて通信回線を長く占有するタイプのネットワークアプリケーションはうまく動作しない．この場合，アプリケーションの通信処理をRPC 形式から遠隔評価形式[9]に変えることで効果的に対処できる．

実行環境の動的な状態変化は事前に予測できないため，ソフトウェア自身がそうした状況に対処（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，その時々の実行環境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．これまではオペレーティングシステム（OS）レベルでの動的適応可能性が主に研究されてきた．そうした適応可能なOSとしてSPIN[1]，Exokernel[3]，Apertos[13]などがある．しかしOSレベルの適応だけでは十分ではない．先の例のように，各アプリケーションの振舞いを変える必要があることも多く，実行環境の動的な状態変化をアプリケーションレベルで積極的に利用することで得られる利点も少なくない．両者の適応は相補的な関係と位置付けられる．

本研究は，アプリケーションレベルの動的適応可能性の実現を目指しており，その実現に際し，我々の関心は以下の3点に集約される．

- **拡張性，保守性，可読性**の高さ．
- 問題領域の記述からの**分離・独立性**．
- 既存言語によるソフトウェアへの**親和性**．

プログラマが事前にソフトウェアの実行環境や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．そしてソフトウェアの記述において，そのソフトウェアが本来目的とする問題領域の記述と動的適応可能性のための記述とが混在するアドホックな実装では，ソフトウェアの可読性を低下させ，その保守と拡張を困難にする．また既存のソフトウェア資産を開放型分散システム環境や移動計算機環境で有効に利用したいという要求は大きい．

本研究では，主として手続きを基本とする動的適応可能性と**メタレベルアーキテクチャ**[6]を用いて，上記の要求を満たす動的適応可能なソフトウェアの実現を目指す．また本研究で提案する言語LEADを用いることで，こうした動的適応可能なソフトウェアを構成することが可能となる．

本稿の構成は次のとおりである．2章では，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基本概念とその利点，およびその実現に要求される機構について述べる．3章では，LEADによる動的適応可能なソフトウェアの構造を示す．4章では，LEADの主要な機能について述べ，5章でLEADの実行支援環境について述べる．6章ではLEADによるプログラミングを具体例を用いて示す．7章で関連研究とLEADとの比較を行い，最後に8章で本稿をまとめる．

2. 動的適応可能なソフトウェアモデル

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**総称手続き**に基づく．総称手続きは複数の**メソッド**から構成され，個々のメソッドは実行コードと実行環境の状態に対する条件（ディスパッチ条件）の組として構成される．総称手続きが呼ばれると，その時々の実行環境の状態に合致する条件を持つメソッドが動的に選択され，実行される．この実行環境の状態に適したメソッドの選択はディスパッチモジュールによってなされ，メソッドの選択方針を**適応の戦略**と呼ぶ．
- 総称手続きおよびメソッドは他に影響を与えない独立性の高いモジュールとして実現される．
- 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．メタレベルは，総称手続き（メソッドを含む），適応の戦略，およびディスパッチモジュールから構成され，ベースレベル（問題領域の記述）から分離・独立される．

このような総称手続きに基づく動的適応可能性は，以下の利点を持つ（**図1**）．

- 手続きをベースに動的適応可能性を実現できる（手続き呼び出しの機能を持つ既存言語によるソフトウェアへの適用が原理的に可能）．
- 総称手続きおよびメソッドの修正・追加によりソフトウェアの動的適応可能性を向上できる（こうした追加・修正は，他の総称手続きやメソッドに影響を与えることなく，安全に行うことができる）．
- メソッドの実行コードを増減することで，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

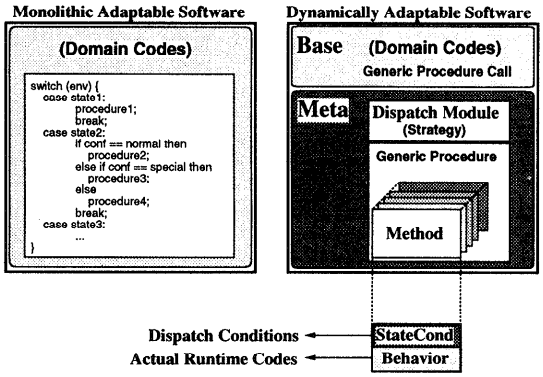


図1 動的適応可能なソフトウェアモデル
Fig. 1 Dynamically adaptable software model.

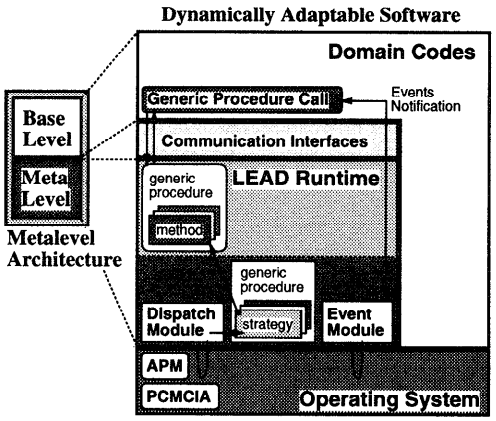


図2 動的適応可能なソフトウェアの構造
Fig. 2 The architecture of dynamically adaptable software.

現できる．

- 適応の戦略を変えることで，総称手続きの柔軟な動的適応可能性を実現できる．
- 動的適応可能性の制御機構が単純に実現でき，その制御にかかるコストを軽減できる．

また，メタレベルアーキテクチャから得られる利点として，以下があげられる（図1）．

- 単層構造の動的適応可能なソフトウェアに比べて，ソフトウェアの可読性・保守性を高める．
- ベースレベルに影響を与えることなくメタレベルの拡張を安全に行うことができる．
- メタレベル上にある総称手続き，適応の戦略，ディスパッチモジュールは，ベースレベル上の複数のアプリケーションから共有・再利用が可能となる．

このような総称手続きに基づく動的適応可能性を効果的に実現するには，以下の機構が必要となる．

(1) 実行環境を構成する要素（**環境要素**）の状態を取得する機構．
(2) 抽象度の高い環境要素とその状態の定義機構．
(3) 総称手続き，メソッドを記述する機構．
(4) 適応の戦略を記述する機構．

いうまでもなく環境要素の状態を取得する機構は必須である．そして実行環境の状態は複数の環境要素の状態から構成されるため，複数の環境要素をまとめて1つの実行環境とし，その状態を表現する機構が必要である．この機構によりプログラマから低レベルの実行環境とその状態を隠蔽し，可読性の高い簡潔なプログラムの記述が可能になる．また総称手続きやメソッドの記述を支援する機構も必要である．そして事前に実行環境の状態を特定できない以上，すべての状態に対応するメソッドを用意することは不可能であり，状態とメソッドを対応付ける戦略が必須となる．総称手続きに基づく動的適応可能性の実現において適応の戦略をいかに実現するかが大きな鍵となる．しかし汎用的な戦略を実現することは困難であり，かつ柔軟な動的適応可能性を実現するには，ユーザが戦略をカスタマイズできる機構を提供する必要がある．

LEADはこれらの機構を備えた言語であり，かつその実行時システムはメソッドの選択を行うディスパッチモジュールを持つ．

3. 動的適応可能なソフトウェアの構造

LEADによる動的適応可能なソフトウェアは，問題領域の記述（ベースレベル）とLEADの実行時システム（メタレベル）上にある総称手続き（メソッドを含む），適応の戦略，そしてディスパッチモジュールから構成される（図**2**）．ベースレベル中で実行環境の状態に応じて変化すべき処理は，総称手続きとそのメソッドとしてLEADを用いてメタレベル上に定義される．LEADの実行支援環境には，メタレベルとベースレベルが通信するためのインタフェースが用意されており，ベースレベルはこのインタフェースを介してメタレベル上にある総称手続きの呼び出しを行う．ベースレベルから総称手続きの要求を受けると，LEADの実行時システムはOSからその時点の実行環境の状態情報を取得し，ディスパッチモジュールを用いてこの状態に適したメソッドを選択する．メソッドの選択方針はディスパッチモジュールが使用する適応の戦略に基づく．そして選択されたメソッドの実行コード部分がインタフェースを通じてベースレベルに送られ，ベースレベル上で実行される．この一連の処理は通信インタフェースを用いた総称手続き呼び出しの実行にまとめられている．

このような総称手続きだけでは，実行環境の状態変

表1 ハードウェア構成に関する環境要素（一部）
Table 1 Part of environmental elements for hardware configurations.

環境要素（組込み関数）	状態（返り値）
network	ネットワークの接続方式
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

化をトリガとして振舞いを変える（状態変化駆動型の）動的適応可能なソフトウェアを実現することは困難である．そのためLEADは環境要素の状態変化を表す**非同期イベント機能**を持つ．イベント名は環境要素名に対応付けされており，その状態変化の発生を表す．非同期イベント機能は総称手続きと対にして使用される．各アプリケーションは必要なイベントとそれに対する総称手続きをLEADの実行時システムに指定する．指定したイベントに対応する環境要素の状態に変化が発生したとき，LEADの実行時システムは，アプリケーションにその発生を非同期イベントとして通知する．通知を受けたアプリケーション側ではそのイベントに対応する総称手続き呼び出しが自動的になされる．この一連の処理も通信インタフェースによる非同期イベントとその総称手続きの指定にまとめられている．非同期イベント機能により状態変化駆動型の動的適応可能なソフトウェアも効率良く，かつ簡潔に実装できるようになる．またイベントに対する動作として総称手続きを利用することにより不要なイベントのフィルタリングを実現することができる．

4. LEADの主要な機能

4.1 環境要素

LEADは環境要素の状態を取得する組込み関数を持つ．これらの関数はその名前に対応する環境要素を表現しており，各関数は対応する環境要素の状態を値（数値）として返す（**表1**）．これらを使用することで環境要素の状態を取得することができる．

4.2 環境要素と状態の定義

LEADの提供する環境要素（組込み関数）は抽象度が低く，かつ実行環境は複数の環境要素から構成されるため，これらを組み合わせてより抽象度が高く扱いやすい環境要素とその状態を定義できる必要がある．LEADにおけるこの定義例を擬似コードにより以下に示す．下記の擬似コードでは，環境要素 `memory` と `battery` をもとに新しい環境要素 `emergency`（緊急）とその状態（True/False）を定義している．このようなユーザ定義の環境要素とその状態はLEADの実行時システムに登録することで再利用できる．

```
emergency: バッテリが5%以下でメモリが1MB以下
{
    True = 1
    False = 0
    if battery <= 5(%) and memory <= 1(MB)
    then True
    else False
}
```

⟨総称手続き⟩	::=	(⟨Head⟩⟨E-list⟩)
⟨Head⟩	::=	(⟨Name⟩⟨Lang⟩⟨App⟩)
⟨Name⟩	::=	⟨総称手続き名⟩
⟨Lang⟩	::=	⟨アプリケーション記述言語名⟩
⟨App⟩	::=	⟨アプリケーション名⟩|`Generic`
⟨E-list⟩	::=	({⟨環境要素⟩}*)

図3 総称手続きの構成要素
Fig. 3 The constructs of generic procedure.

⟨メソッド⟩	::=	(⟨Head⟩⟨S-list⟩⟨Loc⟩⟨Proc⟩)
⟨Head⟩	::=	(⟨Name⟩⟨Lang⟩⟨App⟩)
⟨Name⟩	::=	⟨総称手続き名⟩
⟨Lang⟩	::=	⟨アプリケーション記述言語名⟩
⟨App⟩	::=	⟨アプリケーション名⟩|`Generic`
⟨S-list⟩	::=	({⟨環境要素の状態⟩}*)
⟨Loc⟩	::=	⟨ホスト名⟩|`local`|`remote`
⟨Proc⟩	::=	⟨実行コード⟩|⟨手続き名⟩

図4 メソッドの構成要素
Fig. 4 The constructs of method.

4.3 総称手続きとメソッド

総称手続きは**図3**の項目から構成される．LEADはさまざまな既存言語で記述されたアプリケーションに動的適応可能性を導入することができる．そのためアプリケーション名とその記述言語名を指定する必要がある．アプリケーション名として `Generic` を指定した場合，その総称手続きは同一言語により記述された複数のアプリケーション間で共有できる．E-listは1つの総称手続きに属する複数のメソッドから実行環境の状態に適したものを選択する際に使用される環境要素のリストである（5.1.2項参照）．

また，メソッドは**図4**の項目から構成される．総称手続きとそのメソッドの定義において，Name，Lang，Appは一致している必要がある．S-listはメソッドのディスパッチ条件であり，実行環境の状態を記述したリストである．具体的には，メソッドが属する総称手続きの定義で指定した環境要素に対する条件式のリストである．S-list中の各環境要素に対する条件式が，実行時にLEADのディスパッチモジュールにより評価され，その結果に基づいて実行環境の状態に適したメソッドが選択される．この条件式の評価方法は適応の戦略に基づいて行われる．

Locは選択されたメソッドの実行コードが実際に実

行されるホストの位置を指定するものである．選択されたメソッドのLocが`local`である場合，そのメソッドの実行コードはユーザが使用している手元のホスト上で実行され，`remote`である場合，デフォルトの遠隔ホスト上で実行される．Locに直接ホスト名を指定することもできる．このメソッドの**リモート実行機能**は，1章で述べたネットワーク通信の接続方式に応じてアプリケーションの処理をRPC形式から遠隔評価形式へ変換する際に有効である．

Procにはメソッドの本体であるアプリケーションの実行コードを指定する．この指定方法として，*1*) 実行コードを文字列として記述する，*2*) 手続きの名前を記述する，の2通りがある．

4.4 適応の戦略

メソッドの選択は各メソッドの持つS-listと適応の戦略によって実現される．LEADのディスパッチモジュールは，適応の戦略を用いて各メソッドのS-listを評価し，適切なメソッドを決定する．この評価方法は，各メソッドのS-list中で使用される環境要素の数やそれらの間の関係などに依存する可能性がある．また最適な戦略が個々の総称手続きごとに異なる可能性もありうる．以上から汎用的な戦略を実現することは難しいうえ，単一の戦略では柔軟な動的適応可能性を実現することが不可能であるため，適応の戦略は利用者が適宜，変更できるだけでなく，容易かつ安全に拡張できる必要がある．このためLEADの実行時システムでは，適応の戦略を総称手続きとして実現し，拡張可能な方式で提供している．

適応の戦略を実現する総称手続きは個々の戦略をメソッドとして持ち，このメソッド（戦略）のディスパッチは，LEADの実行時システム中に定義されている大域変数の値（戦略名を表す文字列）によって行われる．しかし，通常の総称手続きと同様に，何らかの実行環境の状態に応じてディスパッチされるように変更することも可能である．また各アプリケーションは，通信インタフェースを通じて適宜戦略の変更を行うことができる．

総称手続きの動作を規定するディスパッチモジュールは，総称手続きに対するメタレベルと見なすことができる．しかしそのディスパッチモジュールの動作を規定しいてる適応の戦略自体もやはり総称手続きとして実現されている．そしてメタレベル上の総称手続き（正確には，メソッド）はベースレベルの記述（実行コード）を含んでいるため，総称手続き中に通信インタフェースを用いた戦略の変更を記述することができる．つまり総称手続きの動作により，それを規定しているディスパッチモジュールの動作を実行環境の状態に応じて動的に変更することができる．ディスパッチモジュールの動作を変更（戦略を変更）することは，総称手続きの動作に必然的（因果的）に影響する．

計算システムが自己の構成や計算過程に関する計算を行うことを**自己反映**[6),11)]と呼び，自己反映性を持つシステムは自己の構成や計算過程を柔軟に変更可能とする．つまり，上記のような戦略の変更にともなう動作は自己反映的な動作と考えることができ，LEADによる動的適応可能なソフトウェアは自己反映性を持つと考えられる．この自己反映性を利用して，実行時にソフトウェアの機能の一時的な制限やその解除といった柔軟な適応動作を実現することができる．

また，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，事前に定義できるメソッドも自ずと限定される．戦略を変更することで，限られた有限のメソッドを有効に活用し，動的適応可能性を高めることが可能になる．

5. 実行支援環境

LEADの実行支援環境は，*1*) LEADの実行時システム，*2*) 通信インタフェース，から構成される．

5.1 組込みの戦略

LEADの実行時システムが提供する機構で重要なものは適応の戦略である．LEADの実行時システムにあらかじめ用意された適応の戦略には**単純戦略**と**優先度数戦略**がある．ソフトウェアの動的な適応動作には，*1*) ある状態においてのみ行う必要のある特別な適応動作と，*2*) すべての状態において行う適応動作，の2通りが考えられる．単純戦略は，*1*のような特別な適応動作の実現のため，優先度数戦略は，*2*のような通常の適応動作を実現するためにそれぞれ提供されている．つまり前者はある状態ではメソッドの選択をせず，処理を行わないが，後者はいかなる状態のときも何らかのメソッドを選択し必ず実行する．これらの戦略は1つの総称手続き中のメソッドとして実現されている．ユーザは新たな戦略をメソッドとして追加することも，これらの組込みの戦略を実現するメソッドを変更することもできる．

5.1.1 単純戦略

単純戦略にはAND戦略とOR戦略がある．前者は各メソッドのS-list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S-list中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

環境要素の数，指定位置	=	$n,\ k$
環境要素の優先度数	=	2^{n-k}
E-list 中の環境要素	=	$(E_1 \ldots E_n)$
優先度数リスト	=	$((E_1 \cdot 2^{n-1}) \ldots (E_n \cdot 2^0))$
環境要素とその状態	=	$((E_1 \cdot S_1) \cdots (E_n \cdot S_n))$
正の和	=	$\sum_{i=1}^{n} 2^{n-i} \cdot \lfloor \varepsilon(S_i) \rfloor$
負の和	=	$\sum_{i=1}^{n} 2^{n-i} \cdot (1 - \lceil \varepsilon(S_i) \rceil)$

図5 優先度数とその計算
Fig. 5 The priority number and its calculation.

選択する．もし条件を満たすメソッドが存在しないならば，単純戦略はメソッドの選択を行わず，総称手続きの実行は行われない．また，単純戦略では，複数のメソッドが選択される可能性がある．その場合，最後に定義されたメソッド（最も新しいメソッド定義）を選択する．

5.1.2 優先度数戦略

優先度数戦略は総称手続きの定義で指定される環境要素間に優先順位を付け，この優先順位と各環境要素に対する条件（メソッドの S-list）の評価結果（真偽）を考慮してメソッドを選択する．総称手続きの定義で指定される各環境要素には，その指定順（左から右）に基づく優先度数が付けられる（**図5**）．LEAD の実行時システムは総称手続きの定義時に，各環境要素とその優先度数の組からなる優先度数リストを総称手続きごとに生成する．そしてメソッドの S-list 中の条件（各環境要素の状態に対する条件）を順に評価する．この評価関数 ε は，環境要素 E_i の状態 S_i が条件を満たす場合，$\varepsilon(S_i) = 1$，条件を満たさない場合，$\varepsilon(S_i) = 0$ となる．また環境要素 E_i が未定義のとき，$\varepsilon(S_i) = 0.5$ となる．この関数と各環境要素の優先度数を組み合わせることにより優先度数の**負の和**および**正の和**を算出する（図5）．

さらにメソッド間でこの2つの和を比較し，メソッドを決定する．比較の際，以下の規則を順に適用する．

規則 α：負の和が最小であるメソッドを選択する．

規則 β：正の和が最大であるメソッドを選択する．

まず規則 α を適用し，それでメソッドが決定できなかった場合，規則 β を適用し，メソッドを選択する．メソッドを確実に決定するために各環境要素には2のべき乗の優先度数が付けられている．優先度数戦略はプログラマが総称手続きの定義時に優先順に環境要素の指定を行うことを仮定し，プログラマの指定した優先順位の高い環境要素に対する条件が真となるメソッドを優先的に選択する方針を基本としている．

5.2 通信インタフェース

LEAD の実行支援環境には，LEAD の実行時システムと各アプリケーションが通信を行うためのインタフェース関数が用意されている（**表2**）．これらの関数の実装は各アプリケーションの記述言語ごとに用意されているため，アプリケーション中でこれらは通常の関数呼び出しと同様に実行することができる．これらの関数を使用することで LEAD の実行時システムが提供する機能を利用することが可能となる．

表2 インタフェース関数
Table 2 The interface functions.

インタフェース関数	用途
GenericProc	総称手続き呼び出し
RegistEvent	イベントの指定
EventProc	イベントのための総称手続き指定
ChangeStrategy	適応の戦略の変更

6. LEAD によるプログラミング

我々は Scheme を拡張したインタプリタ型言語として LEAD のプロトタイプを実装した．このプロトタイプが支援する既存言語は Emacs Lisp, Perl, Tcl/Tk, Python であるが，UNIX のソケットに基づく通信機能を提供するインタプリタ型の言語であれば，どれも原理的に支援可能である．プロトタイプ，通信インタフェースおよび例題の実装はすべて BSD/OS 2.0.1 Wildboar☆と SunOS 4.1.4☆☆上で行った．

6.1 例題の仕様

本研究では，例題として GUI ベースの動的適応可能なテキストエディタ（**図6**）とリソースモニタを作成した．これらは，デスクトップパソコン上だけでなく，バッテリ駆動可能なノート型パソコンでの使用も考慮しており，ベースレベル記述言語は Tcl/Tk である．これに LEAD を用いて動的適応可能性の導入を行った．本稿ではテキストエディタの記述のみ示す．

以下は，このテキストエディタの適応可能な動作の仕様である．

- 使用するディスプレイのサイズに応じてウインドウ（ダイアログを含む）や文字のサイズを実行時に変更する．さらにウィジェット（部品）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（メモリの節減と効率的な入出力）．
- 電源の種別に応じてファイルの自動保存の回数を増減させる（バッテリの消費を節減）．

☆ APM（Advanced Power Management）や PC カードサービスを支援している版．

☆☆ APM や PC カードサービスを支援していないため，ハードウェアの構成要素を表す環境要素が制限される．

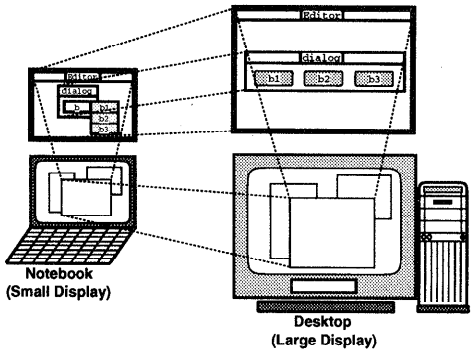


図6 動的適応可能なテキストエディタ
Fig. 6 The dynamically adaptable text editor.

6.2 総称手続きとメソッドの記述

総称手続きとメソッドは LEAD の特殊フォーム **defgeneric** と **defmethod** を用いて定義される．下記のコードは，このエディタのファイル選択ダイアログを実現する総称手続きとメソッド定義の一部である．このダイアログは使用するディスプレイのサイズに応じて変化するため，環境要素として**display** を使用する．各メソッド中では環境要素 **display** に対する条件を指定する．メソッド 1 はサイズの大きなディスプレイの使用時に，メソッド 2 はサイズの小さなディスプレイの使用時に，それぞれディスプレイのサイズに見合ったファイル選択ダイアログを実現する．

```
(defgeneric  ;総称手続きの定義
   FileSelectDialog ;総称手続き名
   tcl        ;アプリケーション記述言語名
   Editor     ;アプリケーション名
  (display)) ;環境要素（ディスプレイサイズ）
(defmethod   ;メソッド 1 の定義
   FileSelectDialog ;総称手続き名
   tcl        ;アプリケーション記述言語名
   Editor     ;アプリケーション名
  ((display >= LARGE)) ;S-list（ディスプレイ大）
   local      ;実行位置
  "proc FileSelectDialog {w} {...}") ;実行コード
(defmethod   ;メソッド 2 の定義
   FileSelectDialog
   tcl
   Editor
  ((display = SMALL)) ;S-list（ディスプレイ小）
   local
  "proc FileSelectDialog {w} {...}")
```

この総称手続きの実行はインタフェース関数を使い，アプリケーション中で以下のように行う．

```
GenericProc "Editor FileSelectDialog $w"
```

インタフェース関数 **GenericProc** の引数はアプリケーション名（**Editor**），総称手続き名，そしてその総称手続きへの引数である．

6.3 非同期イベントの利用

このエディタは電源の種別に応じたファイル自動保存の回数を変化させるため，非同期イベントを利用している．ノート型パソコンの使用時にはユーザが適宜電源を切り替える可能性がある．この変化は予測できないため非同期イベントによりその発生を取得する．

```
(defgeneric ;総称手続きの定義
   AutoSaveTimes ;総称手続き名
   tcl       ;アプリケーション記述言語名
   Editor    ;アプリケーション名
  (power))  ;環境要素（電源種別）
(defmethod  ;メソッド 1 の定義
   AutoSaveTimes ;総称手続き名
   tcl       ;アプリケーション記述言語名
   Editor    ;アプリケーション名
  ((power = AC-POWER)) ;S-list（固定電源使用）
   local     ;実行位置
  "proc AutoSaveTimes {f} {...}") ;実行コード
(defmethod  ;メソッド 2 の定義
   AutoSaveTimes
   tcl
   Editor
  ((power = BATTERY)) ;S-list（バッテリ使用）
   local
  "proc AutoSaveTimes {f} {...}")
```

上記のコードはこの機能を実現する総称手続きとメソッド定義の一部である．メソッド 1 は固定電源使用時に，またメソッド 2 はバッテリ使用時に，それぞれ実行される．ファイル自動保存機能を実現するベースレベルの元々の処理は，Tcl/Tk のキー入力イベントにより起動されるイベント駆動型の手続きとして実装された．これを総称手続き化する際に，非同期イベントを利用することで，電源種別に変化が起きた際にも起動されるようになる．このような非同期イベントは Tcl/Tk では支援されていない．そしてこの総称手続き化にあたり，ファイル自動保存の回数を制御するようにした．具体的には，バッテリ使用時はファイルの自動保存回数を減少させる（ディスクアクセスの減少）ことでバッテリの消費を節約する．この総称手続きの実行はインタフェース関数を使い，アプリケーション中で以下のように行う．まず，**RegistEvent** 関数を用いて電源イベントの指定を行う．この記述により，使用する電源の種別が変わったとき，その変化の発生を 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ようになる．そしてイベント取得後の動作として総称手続きの指定を **EventProc** 関数を用いて行う．この指定によりイベント取得後，指定した総称手続き **AutoSaveTimes** が自動的に実行されるようになる．

```
RegistEvent  "power" #電源イベント
EventProc    "Editor AutoSaveTimes $fl"
```

非同期イベントを使用するか否かの明確な基準はない．しかし，アプリケーションの動作中に，その変化が発生する可能性が高く，かつその変化への対応が早急に必要となるような処理には，非同期イベントの利用が有効である．たとえば，先のファイル選択ダイアログでは非同期イベントを利用していないが，その理由は，ディスプレイのサイズがアプリケーションの実行中に突然変化するような状況はきわめて少ないと考えられるからである．しかし，そういう状況が稀でない実行環境であれば，ファイル選択ダイアログの実現に非同期イベントを用いることは妥当であり，有効である．

6.4 適応の戦略の変更

ファイル自動保存の機能において，バッテリ使用時でかつその残量が半分以上であった場合，ファイル自動保存の回数を減少させ，さらにその残量が半分以下となった場合，ファイルの自動保存を行わず，バッテリの消費を回避したい，という要求も考えられる．これを実現するには前節の定義を次のように修正する．まず，環境要素batteryを総称手続きの定義に追加指定する．

```
(defgeneric ;総称手続きの定義の修正
   AutoSaveTimes ;総称手続き名
   tcl       ;アプリケーション記述言語名
   Editor    ;アプリケーション名
  (power battery)) ;環境要素の追加
```

次に先のメソッド2の定義にバッテリの残量が半分以上（MIDDLE）という条件を以下のように追加する．

```
(defmethod  ;メソッド 2 の定義の修正
   AutoSaveTimes ;総称手続き名
   tcl       ;アプリケーション記述言語名
   Editor    ;アプリケーション名
  ((power = BATTERY)(battery >= MIDDLE)) ;S-list
   local     ;実行位置
  "proc AutoSaveTimes {f} {...}") ;実行コード
```

ここでは，固定電源使用時のメソッド1については特に変更する必要はない．同様に，バッテリ残量の変化を非同期に通知するバッテリイベントの追加指定を以下のように行う．

```
RegistEvent  "power battery" #イベントの追加
```

これまであげたエディタの適応動作はすべて適応の戦略として優先度数戦略を使用している．その理由はさきのエディタの適応動作は，いずれも実行環境のある状態においてのみ行う必要のある特別な動作ではないからである．しかし，本節で修正したファイル自動保存機能は，電源がバッテリでかつその残量が半分以下であった場合以外にのみ行う特別な処理となるため優先度数戦略は適切ではなくなる．優先度数戦略は必ず行うべき適応動作を実現するために，メソッドを必ず選択するからである．それゆえ適応の戦略を優先度数戦略から単純戦略（この場合，AND戦略）に変更する必要がある．戦略の変更はインタフェース関数を使いアプリケーション中で以下のように行う．

```
ChangeStrategy  "and" # AND 戦略への変更
```

この記述によりメタレベル上の適応の戦略がAND戦略に変更され，これ以降，AND戦略によりメソッド選択がなされる．そのため，メソッド中の条件（S-list）を完全に満たすメソッドがない場合，メソッドは選択されず，ファイルの自動保存は行われなくなる．ここで問題となることは，戦略の変更は大域的な影響を及ぼすことである．つまりすべての総称手続きの戦略を変更してしまうことになる．しかし，インタフェース関数はベースレベル記述言語ごとに実装されているので，メソッドの実行コードに戦略の変更処理を記述することができる．つまりメソッド中に戦略の変更を行うコードを記述しておき，実行環境のある状態のときには戦略を単純戦略に変更して，アプリケーションの（部分的な）機能の一時的な制限を行い，それ以外の状態に戻ったときに，また戦略を変更して機能の制限解除を行うようにすることができる．こうした戦略の変更とその局所化により，このような柔軟な動的適応可能性を実現することが可能となる．

6.5 議　　論

本章の例題をLEADを用いずベースレベル記述言語のTcl/Tkだけで実現する場合を考慮すると，LEADの有効性の一端を示すことができる．まず，Tcl/Tkは環境要素の状態を取得する組込み関数がないため，それらを新たに実装する必要がある．また，Tcl/Tkは問題領域の記述から動的適応可能性のための記述を分離する機構がないため，可読性や保守性に対する問題が生じる．さらにソフトウェアの適応動作を共有，再利用，拡張の容易な総称手続きおよびメソッドとして実現する機構もないうえ，適応の戦略を総称手続きとして実現する機構も提供されない．このため拡張の容易かつ安全な動的適応可能性を実現するのは非常に困難となる．そのうえ，Tcl/Tkは環境要素の状態変化を表す非同期イベントがないため，これを擬似的に実現するにはアプリケーションがポーリングして実行環境の状態をつねに監視する必要がある．このような

実装は繁雑かつ効率が悪い．こうした点はTcl/Tkだけにあてはまるものではなく，他の既存言語についても程度の差はあっても同様にいえることである．

しかし本研究で提案する動的適応可能な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．Tcl/Tkを含めた既存言語に本稿であげたソフトウェアの機構を導入し，拡張することで本研究のソフトウェアモデルを実現することができる（実際に，LEADのプロトタイプはSchemeの拡張である）．つまり本研究の手法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．

7. 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いくつかの研究がなされている．Rome[2]やClovers[10]では，オブジェクトがクラスに基づく複数の表現を同時に持つことができ，オブジェクトの表現を動的に変更することができる．しかし，これらの表現の変更方法，つまり適応の戦略は支援されていない．そのうえ，これらの表現の変更には何ら制約がないため，こうした変更を多用することは，プログラムの可読性を損なうだけでなく，その保守を困難にする．適応化コンポジション[4]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，環境自体の動的な変化について考慮されていない．Gaea[7]では，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，環境自体を明示的に表現する機構がない．AL-1/D[8]では，メタオブジェクトがLEADの環境要素に相当するシステム資源の統計情報を持ち，その情報に対する制約を解くことでメソッドを自動的に起動することができる．しかしこの統計情報はシステムによって固定されており，その数も少なく，その抽象度も低い．

LEADでは，メソッド中のS-listが実行環境の状態を表現しており，その意味では，ある程度環境の記述を事前に必要とする．しかしメソッドはS-listと適応の戦略によって決定されるため，S-listの記述が不完全であってもそれを戦略が補うかたちで環境に適切なメソッドを選ぶことができる．戦略を強化することでS-list中の記述を最小限にすることも可能である．また上記の関連研究はLEADのように適応の戦略部分を柔軟にカスタマイズすることができない．さらに，これらのアプローチはその記述言語や特定の応用領域に依存する度合いが強く，それらを他の言語によるソフトウェアに導入しにくいという欠点がある．本研究のアプローチは単純で特定の言語に依存しないため，さまざまな言語に適用することが原理的に可能である．

8. おわりに

本研究では，動的適応可能なソフトウェアのモデルを提案し，それを構成する言語LEADの設計と実装を行った．そしてLEADを用いて例題を作成した．LEADによる動的適応可能なソフトウェアは可読性，保守性に優れ，その総称手続きは高い拡張性と再利用性を有する．またLEADを用いることで，既存のソフトウェア資産を有効かつ発展的に利用することが可能となる．

今後の予定としては，適応の戦略の局所的な変更をシステマティックに実現する機構と構文の導入や，より規模の大きなソフトウェアに動的適応可能性を導入し，その有効性の検証と導入にかかるコストの定量的な評価を行う（この検証方法や評価方法も今後の課題である）などを検討している．また，JavaやTelescriptなどの言語支援も検討している．これらの言語によるモーバイルエージェントは実行される環境を考慮しないで作成されるため，実行環境の状態によってはうまく動作しない可能性がある．本研究の手法により，こうした事態の回避は原理的に可能であると考えている．

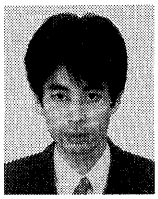
謝辞 本研究を行うにあたって貴重なご助言をいただいた北陸先端科学技術大学院大学の中島達夫，緒方和博の両先生ならびに同大学院言語設計学講座の諸氏に深謝する．また，有益なコメントをいただいた査読者の方々に感謝の意を表する．本研究の一部は情報処理振興事業協会（IPA）「独創的情報処理技術育成事業」の一環として行われたものである．

参考文献

1) Bershad, B.N., Savage, S., Pardyak, P. and Sirer, E.G.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating System Principles* (*SOSP-15*), pp.267–284 (1995).
2) Carre, B., Dekker, L. and Geib, J.: Multiple and Evolutive Representation in the Rome Language, *Proc. Technology of Object-Oriented Language and Systems* (*TOOLS'90*) (1990).
3) Engler, D.R., Kaashoek, M.F. and Jr, J.O.: Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. 15th ACM Symposium on Operating Sys-*

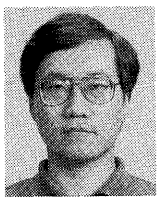
tem Principles (*SOSP-15*) (1995).
4) 本田康晃，渡　滋，所真理雄：適応化コンポジション，コンピュータソフトウェア，Vol.9, No.2, pp.48–62 (1992).
5) Lange, D.B., Oshima, M. and Kosaka, K.: Aglets: Programming Mobile Agents in Java, *Proc. World-Wide Computing and its Applications* (*WWCA'97*), Lecture Note in Computer Science, Vol.1274, pp.253–266, Springer-Verlag (1997).
6) Maes, P. and Nardi, D. (Eds.): *Meta-level Architecture and Reflection*, Elsevier Science Publishers (1988).
7) Nakashima, H.: Organic Programming for Cooperative Computation, *Proc. World-Wide Computing and its Applications* (*WWCA'97*), Lecture Note in Computer Science, Vol.1274, pp.133–141, Springer-Verlag (1997).
8) Okamura, H., Ishikawa, Y. and Tokoro, M.: AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework, *Proc. IMSA'92 International Workshop on Reflection and Meta-level Architecture* (1992).
9) Stamos, J. and Gifford, D.: Remote Evaluation, *Trans. Programming Languages and Systems* (*TOPLAS*), Vol.12, No.4, pp.537–565 (1990).
10) Stein, L.A. and Zdonik, S.B.: Clovers: The Dynamic Behavior of Types and Instances, Technical Report, CS-89-42, Department of Computer Science, Brown University (1989).
11) 渡部卓雄：リフレクション，コンピュータソフトウェア，Vol.11, No.3, pp.5–14 (1994).
12) White, J.E.: Telescript Technology, Mobile Agents, White Paper (1996).
13) Yokote, Y.: The Apertos Reflective Operating System: The Concept and Its Implementation, *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'92*), pp.414–434 (1992).

(平成 9 年 8 月 26 日受付)
(平成 10 年 7 月 3 日採録)



天野　憲樹（学生会員）

1966 年生．1990 年日本大学文理学部哲学科卒業．1996 年北陸先端科学技術大学院大学情報科学研究科情報システム学専攻修士課程修了．現在同専攻博士課程在学中．プログラミング言語，自己反映計算，移動計算機環境に興味を持つ．日本ソフトウェア科学会会員．



渡部　卓雄（正会員）

1963 年生．1986 年東京工業大学理学部情報科学科卒業．1991 年同大学大学院後期博士課程修了．理学博士．1990～1992 年日本学術振興会特別研究員（東京工業大学，東京大学）．1992 年より北陸先端科学技術大学院大学情報科学科助教授．ソフトウェア基礎論，プログラミング言語，並行計算，分散処理に興味を持つ．情報処理学会平成 3 年度研究賞受賞．日本ソフトウェア科学会，ACM 各会員．