移動計算機環境をサポートする言語 LEAD による非同期イベント処理

Asynchronous Event Handling in LEAD: A Language for Mobile Computing Environments

天野憲樹　渡部卓雄*

北陸先端科学技術大学院大学†

概要

LEAD はメタレベルアーキテクチャによりアプリケーション本来の記述とシステムの動的変化に適応するための制御記述をモジュール化する言語である。さらにシステムの状況に応じて変える必要のある処理をアプリケーション本体から分離する。LEAD によりシステム構成や環境の動的変化に適応可能なアプリケーションが容易に作成可能となり、各アプリケーションを構成するモジュールの容易な拡張、共有、再利用が実現される。LEAD の実行時システムはシステムの構成が変化した時、その情報をアプリケーションに非同期イベントとして通知する機能を持つ。各アプリケーションはこの情報をもとにシステムの状況に応じた振舞いを実現することが可能になる。

Abstract

To construct adaptable applications on mobile computing environments, the LEAD language (1) provides portable and extensible mechanism for context awareness, and (2) supports modular description of adaptation policies. By using LEAD, the extension, reuse and share of modules which construct adaptable applications are improved. LEAD runtime has a functionality of handling asynchronous events. When mobile hosts' underlying hardware configurations change dynamically, LEAD runtime notifies applications of the informations asynchronously as events. The applications which receives the events can realize a suitable behavior for the informations of the system configuration by communicating with LEAD runtime.

*Noriki AMANO, Takuo WATANABE, {n-amano, takuo}@jaist.ac.jp

†Graduate School of Information Science,Japan Advanced Institute of Science and Technology

1 はじめに

小型計算機の高性能化、PC カードや無線通信の技術進歩、そしてネットワーク環境の急速な整備と普及は、移動計算機環境の実用性を高めた。しかし真に実用的な移動計算機環境を実現するには多くの課題が残されている。移動計算機環境では、計算機システムの構成やそれを取り巻く環境 (以降、システム環境) が動的に変化する可能性がある (計算機の移動にともなうIP アドレスの変化やPC カードの脱着等)。このような動的に変化するシステム環境に対して、その上位のアプリケーションは何ら対応するすべを持たない。その結果、同じアプリケーションであってもシステム環境によって著しく効率や操作性が低下する場合がある。しかし携帯可能な計算機の出現とネットワークの整備・普及にともない、いつ、どこで、どのようなシステム環境でも、それを意識せずに同じアプリケーションを使用したい、といった要求が高まっている (例えば、メールの読み書き等)。移動計算機環境に対する従来のアプローチは、動的なシステム環境に適応可能な OS によってシステムの動的な変化を吸収し、アプリケーションに対してシステムやネットワークの透過性を実現するものが主であった [3, 4, 5, 9, 11]。

しかしシステム環境の動的変化に対して、OS レベルによる対応だけでは十分とはいえない。というのは移動計算機環境のような環境でシステムの透過性を完全に実現することは不可能であり、さらにシステム環境の変化をアプリケーション側が積極的に利用したい場合もあり得るからである。アプリケーションがこうしたシステムの情報を利用して、その振舞いを変えることで得られる利点は少なくない。このようにシステムの状況に応じてみずからの振舞いを変えるアプリケーションを適応可能なアプリケーションという。本研究ではこうした適応可能なアプリケーションを作成する枠組を提案し、移動計算機上でのアプリケーションの実行を支援する。

この種のシステム環境に応じたアプリケーションの振舞いを全て OS がサポートすることは困難であり、OS が本来提供するべき機能からみて適切とはいえない。これらアプリケーションの振舞いの変更はアプリケーション自身の責任において実現されるべきである。なぜならば、システム環境に応じて振舞いを変えるのはアプリケーションであり、変更の是非や度合はアプリケーションの性質やその処理の性質に依存するからである。しかし、このような処理を実現する機構を各アプリケーションがそれぞれ内部に実装することは現実的ではない。実装にかかる負荷が大きいだけでなく、アプリケーション本来の記述とシステム環境の変化に適応するための制御記述が混在することになり、プログラムの可読性や保守性を低下させる。この種の適応動作のための機構はアプリケーション本体から独立していることが望ましい。またアプリケーション本来の記述中でもシステム環境の変化に関係する部分と関係しない部分を分離して記述できることが望まれる。このモジュール化により多様なシステム環境への拡張性と各モジュールの再利用性が高まる。しかし既存のプログラミング言語はこのようなモジュール化を提供しない。一方、システム環境の変化に適応する機構をライブラリ [7] として実現することも考えられる。しかしこうしたライブラリも個々のアプリケーションの振舞いの詳細な記述に対応するには限界がある。アプリケーションの振舞いはそのアプリケーションが記述されている言語で記述できることが望ましい。

またアプリケーションがシステム環境に応じて振舞いを変化させるためには、適宜システム環境の情報を知る必要がある。このため従来とは正反対のアプローチが必要になる。従来は下位のシステム環境に発生した変化を上位のアプリケーションに隠し、いかにシステムの透過性を実現するかが焦点であった。しかしこの方法ではシステム環境に応じて振舞いを変えるアプリケーションを実現することは不可能である。このため下位のシステム環境が変化した時、その情報をアプリケーションに通知する機構が必要となる。

本研究で提案する LEAD はこれらの点を解決するために設計された言語システムである [1, 2]。LEAD によりアプリケーションプログラムの可読性や保守性を高めることができる。さらにアプリケーション中の状況に応じた振舞いは、LEAD によって独立したモジュールとして構成され、拡張、共有、再利用が容易に可能となる。また LEAD はシステム環境の情報をアプリケーションに提供する機構を二つ持つ。一つはアプリケーションからの要求に応えて、要求されたシステムの状態を返すものであり、もう一つはシステム環境が変化した際、その情報を非同期なイベントとしてアプリケーションに通知するものである。特に後者は今回新たに追加された機能である。この非同期イベントのサポートにより、これまで擬似的に実現していたシステムの状況に応じて受動的に振舞いを変えるアプリケーションが効率的かつ容易に記述可能となった。アプリケーションはこれらの機構を適宜利用することで振舞いを変えることが可能になる。

本稿では、第 2 節でシステムの概要、第 3 節で LEAD が提供する言語の概要を述べる。第 4 節では LEAD の持つ非同期イベント処理の機構について述べ、第 5 節でこの非同期イベント処理を用いたアプリケーションの動作と記述を例題を用いて示す。そして第 6 節で本稿をまとめる。

2 システムの概要

LEAD には二つの側面がある。一つはシステム環境の変化に適応可能なアプリケーションを支援するシステムとしての側面であり、もう一つは適応可能なアプリケーションを支援・作成する言語としての側面である。以降、前者を LEAD ランタイム、後者を LEAD と呼ぶ。LEAD ランタイムは以下のコンセプトに基づいて設計されている。

- アプリケーションに対してメタレベルを提供する。
- システム環境に適応するための制御機構をアプリケーションに提供する。
- アプリケーションとの通信インタフェースを持つ。
- 適応可能なアプリケーションを支援・作成する言語を提供する。
- システム環境の情報をアプリケーションに提供する。

LEAD ランタイムとアプリケーションはメタ/ベースの関係であり、メタレベルアーキテクチャ[8, 12] として構成される (図 1)。メタレベルアーキテクチャとは本来目的とする計算を行うベースレベルとそれを制御するメタレベルの二層構造からなる計算システムの機構である。このようなメタレベルアーキテクチャを用いることで各レベルは分離され、プログラムの可読性、保守性、拡張性さらに再利用性を高めることができる。LEAD ランタイムはアプリケーションにメタレベルとしてそれ自身の一部分を提供する。システム環境に応じたアプリケーションの振舞いはこのメタレベルに記述され、LEAD がそれらの振舞いを制御することで多様なシステム環境の変化に対応可能となる。

また LEAD ランタイムとアプリケーションの関係はサーバ/クライアントの関係でもある。つまり振舞いを変える主体はアプリケーション自身であるが、環境に適した振舞いをディスパッチする機構は各アプリケーションが個々に持つ必要はない。この機構をサーバとしての LEAD ランタイムが提供するため、アプリケーションプログラマは余分な制御コードを書く手間から解放され、本来目的とする振舞いの記述に専念することができる。これはプログラムの可読性と保守性を高める。

このような関係の LEAD ランタイムとアプリケーションは通信インタフェースを持つ。両者が互いに通信することで、動的なシステム環境に適応可能な振舞いが実現される。

そして LEAD ランタイムはシステム環境の変化に適応可能なアプリケーションを支援 作成するための言語 LEAD を提供する。LEAD は幾つかの特徴的な機能を持つ言語である (参照 3 節)。

アプリケーションが振舞いを変えるにあたり、システムの状態を参照する必要がある。アプリケーションや振舞いにはその性質に応じて能動的/受動的アプリケーション (振舞い) の二つが存在し、それらに応じて参照の方法も二とおり用意されている。前者はアプリケーションが必要になった時、LEAD ランタイムに要求を出して、システムの情報を取得することで振舞いを変えるものであり、後者はシステム環境が変化した時に LEAD ランタイムからその情報を非同期のイベントとして受け取り、それによって振舞いを変えるものである。これらは完全に区別すべきものではなく、同一のアプリケーション内に両方の振舞いが共存している場合が多い。LEAD ランタイムはこの両者をサポートする。

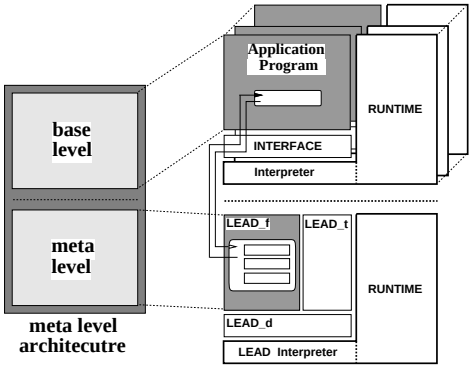

図 1: LEAD のメタレベルアーキテクチャ

3 言語の概要

LEAD は以下のコンセプトに基づいて設計されている。

- 強力な記述力を持つ。
- アプリケーションのモジュール化を促進する。
- LEAD から能動的にアプリケーションに処理を送り実行させる機能を持つ。
- プログラムのリモート実行を提供する。

LEAD は汎用言語と拡張用言語としての両面を持つ。つまり適応可能なアプリケーションを新たにスクラッチから記述することが可能であり、かつ他の言語で作成された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することも可能である。LEAD を用いて、適応可能なアプリケーションを新たに作成する場合、環境に応じて変化させたい振舞いはアプリケーション本体から切り離して LEAD ランタイムの提供するメタレベルに記述する。これは他の言語で作成された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する場合も同様である。つまりシステム環境の変化に関係する部分はアプリケーション本体から完全に分離される。アプリケーションは通信インタフェースをとおして、システムの状況に適した振舞いを LEAD ランタイムに要求する。要求を受けた LEAD ランタイムはディスパッチ機構 [1, 2] により適切な振舞いを選択してアプリケーションに返し、アプリケーション上でその振舞いが実行される。アプリケーション本体は、この振舞いを要求するための記述以外にコードの追加や変更を必要としない。さらにこのメタレベルの記述の一部にはアプリケーションが記述されている言語を使用することができる。このようなメタレベルにシステムの状況に応じた各アプリケーションの振舞いを記述する機能的な側面を LEAD_f と呼ぶ。メタレベルの記述は総称関数とメソッド [10] として表現される。

また LEAD はアプリケーションに対して、システム環境に応じたメソッドのディスパッチ機構を提供するが、この機構も LEAD によって実現されている。このメソッドのディスパッチ機構はいわば LEAD のメタオブジェクトプロトコル [12] としてユーザに公開されているものであり、ユーザは LEAD を用いてディスパッチ機構を容易にカスタマイズすることができる。この機能的側面を LEAD_d と呼ぶ。

このようにアプリケーション中の環境に応じて変化する必要のある部分とそうでない部分をモジュール化することにより、アプリケーションプログラムの可読性、保守性を高め、メタレベルに記述されるシステム環境に応じた振舞いの拡張性、再利用性を高めることが可能になる。

LEAD_f とは逆に LEAD はアプリケーションに対して能動的に働きかける処理が記述可能である。LEAD はアプリケーションを起動し、LEAD_f としてメタレベルに登録されている振舞いのうち、システム環境に適したものを選択し、目的のアプリケーションに送信して実行させ、その結果を受け取ることができる。この機能的側面を LEAD_t と呼ぶ。この LEAD_t を用いて、複数のアプリケーション間に渡る処理をシステム環境に応じて改変する適応可能なスクリプトが容易に実現可能となる。

また LEAD は振舞い単位のリモート実行機能を提供する。この機能によりエージェント指向的なスクリプトを作成し、リモートホスト上で実行することが可能となる。LEAD の提供するリモート実行はシステム環境に応じて自動的にリモート実行を行い、リモートホスト上にないプログラムの実行も可能であるといった利点を持つ。

3.1 総称関数とメソッド

総称関数はシステムの状況に応じた処理の総称であり、異なる振舞いを持つ複数のメソッドへのインタフェースである。総称関数は`defgeneric` 関数を用いて以下のように定義する。

```
(defgeneric 総称関数名
            アプリケーション記述言語名
            (状態要素のリスト))
```

状態要素とはその名前に関連付けられたシステム資源の情報を持つ属性であり、例えばmemory, battery 等がある。状態要素の幾つかは予めシステム中に定義されているが、ユーザが任意に定義することも、再定義も可能である。ここで指定した状態要素の状態に基づいてメソッドが選択される。状態要素は複数指定することができ、その優先度が高い順に左から指定する。

メソッドは個々の状況に応じた処理の実体である。メソッドはdefmethod 関数を用いて以下のように定義する。

```
(defmethod メソッド名
           アプリケーション記述言語名
           (ディスパッチ条件リスト)
           実行ロケーション
           "実行コード")
```

ディスパッチ条件リストとはメソッドが選択される条件を示したリストであり、総称関数で指定した状態要素に対する条件を指定したリストである。複数の状態要素に対する条件を記述することが可能であるが、このメソッドに対する状態要素として総称関数で定義した状態要素に限られる。次に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。また実行コードとはアプリケーション側に送られ実行されるコードであり、アプリケーション記述言語として指定された言語によって記述されるものである。メソッドはLEAD ランタイム中のテーブルに登録される。メソッドのディスパッチはLEAD ランタイムがこのテーブルを参照し、各メソッドのディスパッチ条件を評価することで実現される。LEAD ランタイムはデフォルトのディスパッチポリシを提供している [1, 2]。

4 非同期イベント処理

LEAD ランタイムはシステム環境が変化した時、アプリケーションに対してその情報を非同期なイベントとして通知する機能を提供する (図 2)。これはこれまでのLEAD ランタイム [1, 2] にはない機能であり、今回の拡張はこの実装を主に行われた。本研究のアプローチはアプリケーションの振舞いをシステム環境に応じて積極的に変えることで、移動計算機環境への適応を目指すものである。このためアプリケーションに有効なシステムの情報、あるいは利用可能なシステムの情報を積極的にアプリケーションに通知する。この情報によりシステム環境に適したアプリケーションの振舞いを実現することが可能になる。またこのような非同期なイベントは外界の変化によって受動的に振舞いを変えるアプリケーションの作成を容易にする。各アプリケーションがポーリングしてシステムの状態を監視することなく、システムの状態が変化した時に、非同期にその情報を取得することができる。アプリケーションは以下の手順によりLEAD ランタイムの提供する非同期イベントを容易に利用することができる。

1. 取得するイベントの申請。
2. イベント取得時に起動する総称関数の指定。

アプリケーションはその振舞いを変えるために要するイベントを事前にLEAD ランタイムに申請しておく。イベントは個々のシステムの情報を表し、総称関数の定義で指定した状態要素を指す。これは複数指定することが可能である。次にイベント取得時に起動すべき振舞いを指定する。この振舞いは総称関数としてメタレベルに記述されていなければならない。ここで指定する総称関数も複数指定することができる。なお、状態要素の指定順や総称関数の指定順等の制約は一切ない。どのイベントの発生時にどの総称関数を呼び出すかの指定さえ必要ない。それらの制御は全てLEAD ランタイムが受け持つ。アプリケーション中で必要な指定はこの二つだけである。これらの指定はLEAD ランタイムへの通信インタフェースによって行う。

LEAD ランタイムは申請されたシステムの状態に変化があった場合、それをイベントとしてアプリケーションに通知する。イベントを取得したアプリケーションは指定した総称関数呼び出しを行い、LEAD ランタイムからその時のシステム環境に適したメソッドを取得し、実行する。ここで注意すべき点は、アプリケーションが指定した総称関数が自動的に呼び出されることである。LEAD ランタイムの提供する非同期イベント処理は直観的にはUNIX のシグナルであるが、LEAD ランタイムはアプリケーションに対して低レベルの割り込みを抽象化して隠蔽する。このためアプリケーションプログラマはどのイベント発生時にどの処理を行うかという処理のディスパッチ制御から解放され、本来の振舞いの記述に専念できる。LEAD ランタイムはディスパッチの機構を用いて、現在のシステム環境に合致しないメソッドを除くことができる。つまり間接的なイベントのフィルタリングを実現している。

この非同期イベント処理は総称関数を利用することから、総称関数の再利用性を高める。例えば、あるアプリケーションをシステム環境に応じて能動的に振舞いを変えるアプリケーションとして作成したとする。その後、方針を変えて、そのアプリケーションを受動的に振舞いを変えるアプリケーションとして再構成しようとした時、当初定義した総称関数は何ら変更することなく、そのまま再利用することができる。これは逆のケースについても当てはまる。

このようにLEAD ランタイムの持つ非同期イベント処理は、アプリケーションにはシステムの状態が変化した、という情報だけを伝え、それに応じた処理の制御はLEAD ランタイムが行うという方式であり、割り込みの隠蔽、イベントの間接的なフィルタリング、総称関数の再利用性の促進等、さまざまな利点を持つ。

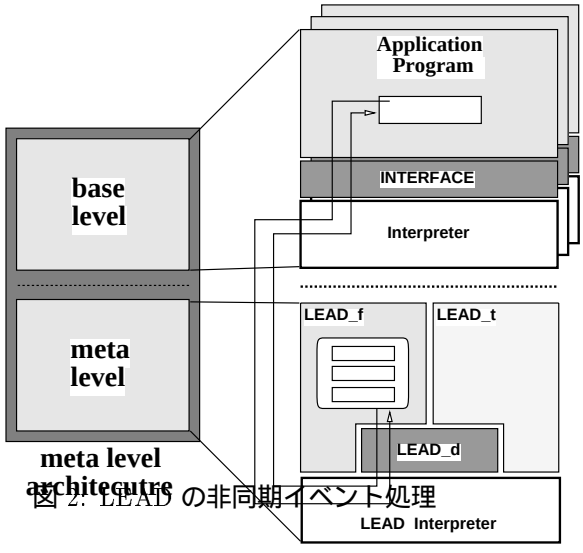

図 2: LEAD の非同期イベント処理

5 例題

本研究では、これまでに作成した適応可能なアプリケーション [2] の振舞いを能動的/受動的という観点から分類して再構成を行った。受動的な振舞いの記述には非同期イベント処理を用いた。これにより各アプリケーションでポーリングすることなく効率的かつ容易な実装が可能となった。LEAD のプロトタイプ、通信インタフェースおよび例題は BSD/OS(Wildboar) 上で実装した。

5.1 適応可能なテキスト ・エディタ

このテキスト ・エディタは能動的に振舞いを変える部分と受動的に振舞いを変える部分の両方が混在しているケースである。このエディタはシステム環境の変化に応じて以下のように振舞いを変える。

1. ディスプレイサイズに応じた振舞い :アプリケーションのメインウインドウ、ダイアログ、文字のサイズを動的に変更する。さらにウィジェットの配置も変更する。ウインドウの大きさやその中のウィジェットの配置等はそのウインドウが開かれる時にディスプレイサイズを取得して決定すればよいので能動的な振舞いとして実現されている。
2. メモリ容量とバッテリ残量に応じた振舞い :文字入出力バッファの伸縮を行う。メモリが十分にある場合は入出力バッファのサイズを大きくとり、効率的な入出力と消費電力の節減に貢献する。メモリが少ない場合は入出力バッファのサイズを抑えて、メモリの消費を回避する。入出力バッファもファイルの操作時に決定されるので能動的な振舞いとして実現される。
3. バッテリ残量に応じた振舞い :ファイルの自動保存回数の増減を行う。バッテリの残量が少ない時には自動保存の回数を抑えて、バッテリの消耗を回避する。この振舞いは徐々に変動するバッテリの値に応じて決定されるものであり受動的な振舞いとして実現される。

以下はテキスト ・エディタのファイル選択ダイアログの振舞いを実現するメタレベル記述の一部である。まず、はじめに総称関数の定義を行い、次にメソッドの定義を行う。はじめのメソッドは 19 インチサイズのディスプレイの時、その大きさに見合った大きさのダイアログとウィジェットの配置を実現し、その次のメソッドは 9 インチサイズのディスプレイ使用時に開かれるダイアログである。

```
(defgeneric FileSelectDialog ; 総称関数名
            tcl ; アプリケーション記述言語名
            (display)) ; 状態要素
(defmethod FileSelectDialog ; メソッド名
   tcl ; アプリケーション記述言語名
   ((display >= 19inch)) ; ディスパッチ条件リスト
   local ; 実行ロケーション
   "proc FileSelectDialog {w} { ; 実行コード
         pack $w.frame -side left -padx 20
         ...
   }" )
(defmethod FileSelectDialog
   tcl
   ((display = 9inch))
   local
   "proc FileSelectDialog {w} {
         pack $w.frame -side right -padx 10
         ...
   }" )
```

この総称関数の呼び出しはアプリケーション中で以下のように行う (ベースレベル記述)。

```
generic_func "FileSelectDialog arg"
```

generic_func はアプリケーションと LEAD ランタイムが通信するためのインタフェースである。

また、以下はファイルの自動保存の振舞いの一部である (メタレベル記述)。先のファイル選択ダイアログの振舞いと同様に総称関数およびメソッドの定義を行う。

```
(defgeneric FileAutoSave ; 総称関数名
            tcl ; アプリケーション記述言語名
            (battery)) ; 状態要素
(defmethod FileAutoSave ; メソッド名
   tcl ; アプリケーション記述言語名
   ((battery < 50)) ; ディスパッチ条件リスト
   local ; 実行ロケーション
   "proc FileAutoSave {arg} { ; 実行コード
         AutoSave $arg 50
         ...
   }" )
(defmethod FileAutoSave
   tcl
   ((battery >= 50))
   local
   "proc FileAutoSave {arg} {
         AutoSave $arg 100
         ...
   }" )
```

はじめのメソッドはバッテリの残量が 50% 以下の時、ファイルの自動保存の回数を減少させるメソッドであり、その次のメソッドはバッテリ残量が 50% 以上の時、ファイルの自動保存の回数を増加させるメソッドを示している。ファイルの自動保存の振舞いは受動的に変化する振舞いであるため、LEAD ランタイムからのイベントの通知によって起動される。アプリケーション中で必要な記述 (ベースレベル記述) は以下である。まずこの振舞いの変更に要するシステムの情報 (状態要素) を LEAD ランタイムに申請し、次にイベントを受け取った時に起動する総称関数の指定を行う。

```
get_event   "battery" ...
eventdriven "FileAutoSave arg" ...
```

実際にバッテリの残量が変化した時、LEAD ランタイムによりその変化を知らせるイベントが通知される。そしてイベント発生時に駆動されるように指定した総称関数が呼び出され、システム環境に適したメソッドが返される。get_event と eventdriven も LEAD ランタイムとの通信インタフェースである。

5.2 適応可能なリソース ・モニタ

システム ・リソースの状態を表示するモニタである。これは受動的に振舞いを変えるアプリケーションの典型であ

り、通常ポーリングが必要である。これまでのLEAD[1, 2]では状況に応じてポーリングする間隔を変え、ポーリングにかかる負荷の緩和を実現したが、これは本質的な解決とはいえない。今回実装された非同期イベント処理により、システム・リソースに変化が生じた場合、LEADランタイムからあげられるイベントを取得し、最新のシステムの状態を表示することができるようになった。アプリケーションはポーリングすることなく、要求したイベントをLEADランタイムから受けることができる。このリソース・モニタは先のテキスト・エディタのファイルの自動保存の場合と同様な手続きをとって実現される。

5.3 評価

このようなシステム環境に応じた振舞いをアプリケーションに付加するのにかかる労力はLEADを用いることによってかなり軽減される。それはLEADランタイムがシステム環境の変化に対する適応動作のための制御機構や非同期イベント処理を提供しているため、アプリケーションではそのための制御記述を一切書く必要がないからである。LEADによって記述されるのはシステム環境に応じた振舞いを実現する総称関数とメソッドだけであり、エディタやモニタ全体からみたLEADの記述量（行数）はごくわずかである（表1）。しかもこの例題の作成にかかったLEADの記述量のうちの大半はTcl/Tkである。これはメソッドの実行コード部分が最も記述量が多いためである。

さらに上記のシステム環境に応じた振舞いはすべて独立したモジュールとして定義されており、アプリケーションから分離され、各モジュール単位での再利用や共有が可能である。例えばディスプレイのサイズに応じてウインドウのリサイズやウィジェットの配置を変える振舞いを実現するモジュールは、このエディタ以外のTcl/Tkのアプリケーションに再利用ないし共有することができる。これらのモジュールはアプリケーション本体とリンクする必要はない。つまり環境に応じた振舞いを実現するモジュール群はアプリケーション本体とは独立して単独で作成することができ、アプリケーションにインクリメンタルに付加することができる。

	エディタ	モニタ
Tcl/Tk（エディタの主処理）	1196	78
LEAD（総称関数とメソッド）	85	9
（メソッドの実行コード）	(65)	(6)

表1: 例題におけるTcl/TkとLEADの記述量

6 まとめ

移動計算機環境におけるシステム環境の変化に適応可能なアプリケーションを作成・支援するために有効な言語LEADを提案した。例題をとおして能動的、受動的に振舞いを変えるアプリケーションの両方をLEADの枠組で効果的に実現可能であることを示した。そしてこの例題の作成からLEADにより既存のアプリケーションに容易にシステム環境の変化に応じた振舞いを付加することができること、プログラムのモジュール性、可読性、保守性、拡張性そして再利用性が高まることを示した。

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] 天野 憲樹, 渡部 卓雄. 移動計算機環境を支援する言語LEADとそのメタレベルアーキテクチャ, Research Report, IS-RR-96-0014S, School of Information Science, Japan Advanced Institute of Science and Technology, May, 1996. http://www.jaist.ac.jp/~n-amano/paper.html

[2] 天野 憲樹, 渡部 卓雄. メタレベルアーキテクチャによる移動計算機環境のための言語 情報処理学会 第1回 モーバイルコンピューティング研究グループ研究会, pp. 9–14, July, 1996.

[3] Bershad, B., Savage, S., Pardyak, P., Sirer, E.G., Becker, D., Fiuczynski, M., Chambers, C., Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System, In Proceedings of SOSP-15, Copper Mountain, CO. pp. 267–284.

[4] Cheriton, D.R., Duda, K.J. A Caching Model of Operating System Kernel Functionality, In Proceedings of the First Symposium on Operating Systems Design and Implementation, Usenix Association, November, 1994.

[5] Ford, B., Hibler, M. Fluke: Flexible μ-kernel Environment Application Programming Interface Reference, The Flux Project Department of Computer Science University of Utah, May, 1996.

[6] Forman, G.H., Zahorjan, J. The Challenges of Mobile Computing, IEEE Computer, Vol.27, No.4, 1994.

[7] Maeda, C., Bershad, B.N. Service without Servers, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

[8] Maes, P. Computational Reflection (Ph. D. Thesis), technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussels, 1987.

[9] Montz, A.B., Mosberger, D., O'Malley, S.W., Peterson, L.L., Proebsting, T.A., Hartman, J.H. Scout: A Communications-Oriented Operating System, Department of Computer Science The University of Arizona Tucson, AZ 85721.

[10] Steele, G. Common Lisp: The Language, Second Edition, Digital Press, 1990.

[11] Tokuda, H., Nakajima, T., Rao, P. Real-Time Mach: Towards a Predictable Real-Time System, In Proceedings of the USENIX Mach Workshop, October, 1990.

[12] 渡部 卓雄. リフレクション, コンピュータソフトウエア, Vol. 11, No. 3, 日本ソフトウエア科学会, 1994.