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

A Language with Meta-level Architecture for Mobile Computing Environment

天野憲樹　渡部卓雄*

北陸先端科学技術大学院大学†

概要

本研究では移動計算機環境におけるシステム構成の動的な変化に柔軟に対応可能なアプリケーションを作成・支援するためにメタレベルアーキテクチャを用いた言語処理系を提案する。本研究のアプローチは、従来のようなOSレベルによるシステム透過性の実現を目指すものではなく、アプリケーションの振舞いをシステム構成に応じて積極的に変化させることで対応するものである。またこの処理系自身もアプリケーションの要求に応じて自らの振舞いを改変し、それがアプリケーションの振舞いに反映する。つまりこの言語処理系とそのアプリケーションは自己反映的な関係にあり、互いの振舞いに影響を与え合うことで、より状況に適した振舞いを実現する。

Abstact

In a mobile (or nomadic) computing system, its underlying hardware configuration (network connections, power sources, etc.) changes dynamically. This fact implies that it is generally difficult to support a transparent/constant view of its operating environment. Thus, to get the full advantage from a mobile computing system, applications on top of it should be aware of its changes and should adapt themselves dynamically. In this paper, we propose a language named LEAD which supports the development of such applications. LEAD is an object-based language with an open, customizable metalevel architecture.

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

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

はじめに

近年計算機は小型かつ高性能になり、加えてネットワークの普及と無線通信技術の進歩により携帯可能な小型計算機の実用性が高まってきた。このような移動可能な小型計算機をネットワークに接続し、分散環境と融合させた計算機環境を移動計算機環境 [4] という。このような計算機環境ではシステム構成や環境が動的に変化する可能性がある。これまでそうした変化に対し、OS 以下のレイヤで変化に対応することが一般的であった。しかしアプリケーションより下位のレイヤで論理的なシステムの透過性を実現しても、移動計算機環境に内在する物理的な制約から解放されるわけではない。このような問題に対処する枠組の一つとして、アプリケーションそのものがシステム構成や環境に応じて振舞いを変えることが考えられる。状況に適した処理を行うことで、内在する制約の緩和に貢献することも期待できる。状況に応じて自らの振舞いを変えるソフトウエアの実現にはリフレクションやメタレベルアーキテクチャ [11, 13] といった技術の応用が可能である。

1 本研究のアプローチ

移動計算機環境に対する従来のアプローチはアプリケーション以下のレイヤでシステム構成・環境の相違を吸収し、システム透過性の実現を目指すものが多い。しかし多様な環境に対してアプリケーションをすべて同一に実行することが望ましいかどうかは疑問の余地が残る。つまりシステムの構成や環境に応じてアプリケーションの振舞いを変えたほうが望ましい場合もあると思える。例えば、ネットワークアプリケーションの使用に際して、接続の媒体がイーサネットである場合と無線通信の場合とでは転送速度が大きく異なり、その実行効率に大きな差がでることは言うまでもない。このような環境では論理的なシステムの透過性を実現してもアプリケーションの効率的実行は望めない。そこで本研究ではアプリケーションの振舞いをシステム構成や環境に応じて適宜切替えるというアプローチをとる。勿論、アプリケーションの振舞いを変えたからといって、内在する物理的制約は解消されない。しかし計算機の使用目的、使用形態に適した処理を適宜選択、実行することにより、内在する制約を緩和することは可能である。

システム構成や環境に応じたアプリケーションの振舞いの実現には幾つかの方法が考えられる。まず、OS レベルやそれに近いレイヤでこの種の機構を実現することが考えられるが、OS にかかる負荷の増大と言う点以外に、OS 本来の役割として、個々のアプリケーションの細かい振舞いまでサポートするというのは適当とは言えない。振舞いの変更はアプリケーションの性質、処理の性質に依存するものであり、振舞いを変える主体はアプリケーションでなければならない。更に OS に組み込むことはカスタマイズを困難にする。このような機構を共有ライブラリとして実装することも考えられる。しかし OS レベルでサポートする場合と同様、個々のアプリケーションの状況に適した振舞いを完全にサポートするのは不可能である。また各アプリケーション毎にそのような仕組みを内部に実装することも考えられる。しかしこれは実装にかかる負荷が大きいだけでなく、アプリケーションが本来目的とする処理コード中に、状況に依存する処理の制御コードが混在することになり、プログラムの可読性、保守性に問題が生じる。これは既存の言語がシステム構成や環境の変化に依存する部分と依存しない部分をモジュール化するように設計されていないからである。以上の点を考慮し、本研究では上記のモジュール化を促進する言語を新たに提案する。この言語の処理系はインタプリタとして実装されており、アプリケーションの振舞いを状況に応じて選択し実行させるサーバとして捉えられる。このサーバは各アプリケーションに対して、メタレベルを提供する。このメタレベルの中に各アプリケーションの処理を記述（各アプリケーションの記述言語で記述する。）し、それをシステム構成・環境に応じてこのサーバが選択し、アプリケーションに実行させる。つまり固定的な処理ではなく、メタレベルを提供することで、通常のサーバよりも多様な環境に柔軟に対応可能となる。

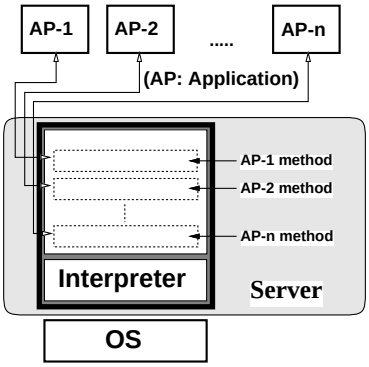

図 1: 本研究のアプローチ

2 プログラミング言語 LEAD

2.1 言語の概要

本研究で提案する言語を **LEAD**[1] と名付ける。LEAD は Scheme に範をとったインタプリタとして実現され、他のアプリケーションプログラムとやりとりするために、socket ベースのプロセス間通信機能を備えている。LEAD は言語として三つの側面を有する。各側面に便宜的に名称を付けることにする。

1. アプリケーション中のシステム構成・環境に依存する処理を記述する。(LEAD_f[2])
2. LEAD インタプリタからアプリケーションを起動し実行させる処理を記述する。(LEAD_t[3])
3. システム構成・環境に依存する処理の制御（ディスパッチ）を記述する。(LEAD_d[4])

つまり LEAD_f とはシステム構成や環境に応じたアプリケーションの異なる振舞いを記述するものであり、LEAD インタプリタ中に登録するものである。これらの振舞いは総称関数（同一名称であり、かつ異なる振舞いをする処理（メソッド）の総称）として定義される。各アプリケーションは LEAD インタプリタへのインターフェースを通して

[1] LEAD のプロトタイプは BSD/OS、SUN-OS 上に C 言語を用いて実装した。
[2] アプリケーションから (from) 要求される処理。
[3] アプリケーションへ (to) 要求する処理。
[4] 状況に依存する処理のディスパッチ (dispatch)。

LEAD インタプリタに対し、総称関数の呼び出しを要求する。そして LEAD インタプリタがシステム構成 ・環境に応じたメソッドをアプリケーションに返し、各アプリケーション上でメソッドが実行される。

次に LEAD_t とは LEAD インタプリタから能動的に各アプリケーションに働きかける処理を記述する。LEAD インタプリタはアプリケーションを起動し、先の LEAD_f として LEAD インタプリタ中に登録されているメソッドを目的のアプリケーションに送って実行させ、その結果を受け取ることができる。

そして LEAD_d は LEAD インタプリタ中に登録された各アプリケーションのメソッドをシステム構成 ・環境に応じて選択するディスパッチ機構を記述するもので、やはり LEAD によって記述されている。メソッドのディスパッチ機構は言わば LEAD のメタオブジェクトプロトコル [13] としてユーザーに公開されているものであり、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。またメソッドのディスパッチは後述のポリシーに基づいて実現され、アプリケーションからの要求により動的に改変することが可能である。

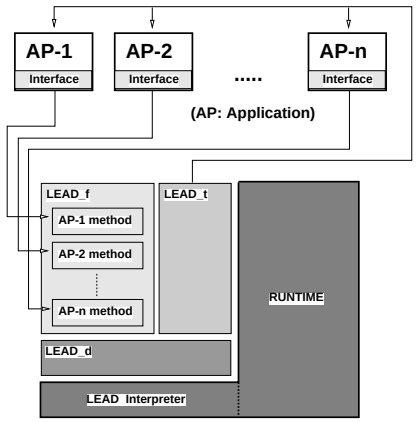

図 2: LEAD とアプリケーション

LEAD インタプリタがサポートするアプリケーションはインタプリタ型の言語で記述されたアプリケーションであり、かつアプリケーションを動かす実行系が socket ベースのプロセス間通信機能を備えていることを想定している。そして LEAD インタプリタとアプリケーションとの通信は socket を用いた双方向のプロセス間通信で実現されている。このためアプリケーション側に LEAD との通信インターフェースが必要になる。本研究では Emacs Lisp、Perl、Tcl/Tk、Python の各言語処理系に対する LEAD とのインターフェースを実装した。

2.2 メタレベルアーキテクチャ

本研究で提案する LEAD とアプリケーションはメタレベル、ベースレベルの 2 層構造で表現され、メタレベルアーキテクチャとして構成される。各レベルは以下のように定義される。

- ベースレベル = アプリケーション
- メタレベル = LEAD インタプリタ

つまりベースレベルであるアプリケーション中の一部分 (システム構成や環境に応じて改変したい処理) を総称関数としてメタレベルである LEAD インタプリタ上で記述し分ける。具体的には、総称関数のうち異なる振舞いをする個々の処理をメソッドという単位で LEAD インタプリタに登録 (定義) する。LEAD インタプリタは、そのディスパッチ機構により、その時々のシステム構成や環境情報に基づいて、複数のメソッドから 1 つを選択し、そのメソッドをアプリケーションに返す。このようなメタレベルアーキテクチャとして構成することでシステム構成や環境に応じたアプリケーションの異なる振舞いをサポートすることが可能になる。

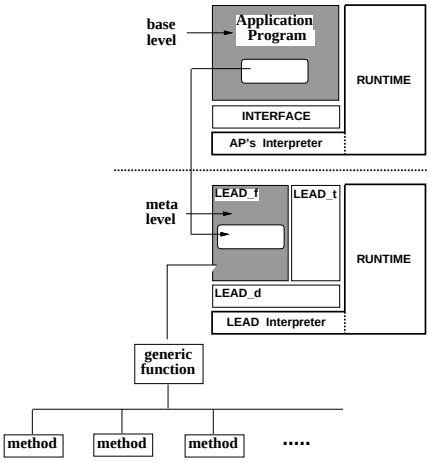

図 3: メタレベルアーキテクチャ

2.3 総称関数

総称関数はある処理を行なうメソッドの集合の総称であり、ある処理について個々のメソッドを記述する前に、その処理が総称関数であることを定義しなければならない。総称関数の定義は **defgeneric** 関数を用いて以下のように行う。

```
(defgeneric 'method-name
            'app-lang
            '(stat-1 ... stat-n))
 method-name ; 総称関数名
 app-lang    ; アプリケーション記述言語名
 (stat-1 ... stat-n) ; 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一種の予約語であり、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、ユーザー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。この関数の実行により、状態要素の優先度を示す優先度数リストが生成される。優先度数は指定された状態要素の数を n とすると、$2^n -$状態要素の優先順位で算出される。

2.4 メソッド

メソッドはアプリケーション中の状況に依存した異なる振舞いであり、**defmethod** 関数を用いて以下のように定義する。

```
(defmethod 'method-name
           'app-lang
           '(cond-1 ... cond-n)
           "host-name"
           "source-code")
 method-name ; メソッド名
 app-lang    ; アプリケーション記述言語名
 (cond-1 ... cond-n) ; ディスパッチ条件リスト
 host-name   ; 実行ホスト名
 source-code ; 実行ソースコード
```

defmethod 関数の実行により生成されたデータはメソッドを登録するテーブルにアプリケーション単位、総称関数単位に区分けされ登録される。メソッドのディスパッチはこのテーブルを参照することによりなされる。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、defgeneric 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。ここで使用可能な状態要素はこのメソッドに対する状態要素として defgeneric 関数で定義した状態要素に限られる。また複数の条件を記述することも可能である。次に実行ホスト名とは続く実行ソースコードを実際に実行するホストを指定する。また実行ソースコードとはアプリケーション側に送られ実行されるソースコードであり、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。このようなメソッド及び総称関数を LEAD インタプリタに定義する言語としての側面が LEAD_f である。

2.5 メソッドのディスパッチ

LEAD におけるメソッドのディスパッチ機構は defgeneric 関数によって生成された状態要素の優先度数リストと、defmethod 関数によって生成されたリスト (以下、メソッドリスト) を参照・評価することで実現される。このメソッドのディスパッチ機構は LEAD によって実装されており、これは LEAD の一つの側面 LEAD_d を指す。

あるアプリケーションからメソッドの呼び出し要求が出されると、LEAD インタプリタは受け取ったアプリケーション記述言語名とメソッド名をキーとしてテーブル中からメソッドを特定する。その際にメソッドリスト中のディスパッチ条件リストの各条件が評価され、現在のシステム構成に適したメソッドが選択される。LEAD はメソッドを選択する際のポリシーとして標準で 3 つのポリシーを提供している。それらのポリシーは以下の 2 つに大別される。

1. 選択メソッドポリシー (and-policy, or-policy)
2. 必須メソッドポリシー (default-policy)

この二つのポリシーは動的なシステム構成上のソフトウエアに要求される振舞いを反映したものである。メソッドにはある特定のシステム構成の場合にだけ必要なものと、全ての場合に必要なものの 2 つが存在する。選択メソッドポリシーは条件があったメソッドを選択するが、条件があわなかった場合にはメソッドを選択しない。このポリシーの実現として and-policy と or-policy の 2 つがある。これらは直観的に明らかなようにディスパッチ条件リストの全ての条件が真 (and) か、またはすくなくとも 1 つ以上が真 (or) の場合、メソッドが選択される。つまりメソッドは選択されるかされないかの二者択一である。反対に必須メソッドポリシーはどのような条件下でも必ずメソッドを選択する。LEAD はデフォルトのディスパッチポリシーとして必須メソッドポリシーをとる。LEAD の目的はシステム構成に応じた異なる振舞いをサポートすることであり、and や or 条件といった単純な場合分けではメソッドが一意に決定できない場合があるからである。必須メソッドポリシーは以下の 2 つのルールに基づいて実現されている。

(a) 各状態要素に対する条件が偽となるものの合計値 (優先度数の合計を負数にした値) が最大であるものを優先する。

(b) 各状態要素に対する条件が真となるものの合計値 (優先度数の合計) が最大であるものを優先する。

この 2 つのルールの優先順位は **(a)** > **(b)** で、(a) は (b) よりも優先される。つまりルール (a) の値が同値であるメソッドについて、ルール (b) を適用することによりメソッドの特定化を行う。

2.6 自己反映的な動作

LEAD インタプリタとアプリケーションは自己反映的な関係にあり、互いの振舞いの変化が互いの振舞いの改変に影響を与え合う。前節で述べた LEAD のディスパッチ機構は LEAD 自身で記述されており、そのポリシーは総称関数として定義されている。そしてインターフェースを通してアプリケーションから使用するポリシーを LEAD に要求することができる。ポリシーの要求を受けた LEAD は該当するポリシーを実現するメソッドをテーブルから選択し、LEAD 上で評価する。これにより新たに選択されたポリシーがロードされ、これ以降は新しいポリシーに基づいてディスパッチが実行される。つまりポリシーの改変はアプリケーションのメソッド選択に反映する。このようなポリシーの改変を要求する処理を総称関数として定義すると、アプリケーションのメソッド選択と LEAD のディスパッチポリシーのメソッドが交互に循環的に改変を繰り返すメタ・サーキュラな構造を作ることが可能になる。すなわちメソッドとポリシーは完全に独立しており、任意の場所でポリシーの変更ができるとともに、ポリシーが総称関数として定義されていることから、ユーザーは他の総称関数と同様に新しいポリシーを自由に追加することが可能となる。

2.7 LEAD によるアプリケーション実行

LEAD インタプリタからアプリケーションを実行する機能は LEAD_t と名付けた LEAD の一つの側面を指す。この LEAD_t は複数のアプリケーション間に渡る振舞いを状況に応じて改変するシェルスクリプトと捉えられる。つまりシステム構成・環境に応じてアプリケーションを組替えたりする処理を記述する。しかし LEAD_t によるスクリプトは通常のシェルスクリプトよりも一段深いレベルの環境に依存した振舞いを実現可能にする。それは LEAD インタプリタからアプリケーションを実行する際、各アプリケーションに要求する処理は LEAD インタプリタにメソッドとして定義されているものでなければならないからである。すなわち、状況に応じてアプリケーションの組替えと同時に各アプリケーション中の振舞いも改変する。これは状況に適したスクリプトを総称関数として定義することで実現される。

2.8 アプリケーションの効率的実行

LEAD はアプリケーションの実行効率を維持するためにプログラムのリモート実行機能を提供する。プログラムのリモート実行は異なる環境におけるアプリケーションの実行効率上の差の解消を意図している。つまり PDA のような携帯端末でデスクトップ環境と同等なアプリケーションの実行効率を維持したい場合、高い処理能力を持つリモートホストの CPU パワーを得ることでレスポンスを維持することができる。また LEAD におけるプログラムのリモート実行は、既存の RPC と比較して、状況に応じて動的にリモート実行が可能、リモートホスト上にないプログラムの実行も可能といった利点を有する。

3 適応可能なソフトウエア

システム構成の動的な変化に柔軟に適応可能なソフトウエアは大別すると次の二種類に分けられる。

1. ある時点における環境情報を必要とするもの。
2. 常に環境情報を把握する必要があるもの。

つまり上記 1 はプログラム実行のある時点でシステムの情報を必要とするソフトウエアであり、その時点のシステム情報に基づいて振舞いを変えるものである。この種のソフトウエアに対応することは難しくない。LEAD はそれらを容易にサポートする。LEAD はメソッドをディスパッチする時に環境情報を取得するからである。しかし 2 のように常にシステムの情報を必要とし、それに基づいて時々刻々と変化するようなソフトウエアに対応するには、通常二つの方法が考えられる。

1. OS からシステム情報が変化した時にその情報をあげてもらう。
2. アプリケーション側で常にポーリングしてシステム情報を拾う。

上記 1 の問題点は、アプリケーションにシステム情報をアップする機能を OS に付加しなければならないことである。OS によるシステム情報のアップコールを前提としているので、そのような機能のない OS では実現不可能である。さらにアップコールを受けるアプリケーションの中に状況に応じたコードを記述しなければならなくなり、これは状況に依存する処理と本来目的とする処理のモジュール化を低下させることになる。また 2 の問題点はアプリケーションが常にポーリングするため無駄な CPU 時間を消費する可能性があることである。例えばシステム構成が全く変化しなかった場合、ポーリングは完全に無駄な処理となる。またシステム構成の変更のタイミングは予測不可能であるため、一定期間スリープさせながらポーリングした場合、変化に即座に対応できない事態を引き起こす。一見するとこの種のアプリケーションを LEAD でサポートすることは難しく見える。しかしこうしたアプリケーションも LEAD を用いることでサポートすることができる。この種のアプリケーションは基本的にポーリングすることが前提となる。問題は常にポーリングするか否かである。前述したように常にポーリングするのでは効率が悪過ぎるし、一定時間スリープすると変化に即対応できない。LEAD ではこの問題を、スリープする処理を総称関数として定義し状況に応じてこの処理を切替えることで回避する。つまり変化に即対応する必要のない場合はスリープ時間の長いメソッドを選択し、逆の場合はスリープ時間の短いメソッドを選択する。これにより極度の非効率性を回避し、かつ変化への対応を比較的スムーズに行えるようになる。

4 例題

本研究では前章で述べた適応可能なソフトウエアの典型的な二つの例題を作成した。これらの例題はシステム構成や環境の変化が著しい移動計算機環境での使用を想定して作成しており、ハードウエアとしては PCMCIA のスロットを装備したノート型パソコンである。

4.1 テキスト・エディタ

前述したある時点における環境情報によって振舞いを変えるアプリケーションとして LEAD と Tcl/Tk を用いて、システム構成・環境に応じて振舞いを変えるテキスト・エディタを作成した。このエディタ [9] は GUI ベースで実現されるフルスクリーン・エディタであり、システム構成や環境に依存して以下のように振舞いを変える。

ディスプレイのサイズに依存した振舞いとしてアプリケーションのウインドウや文字のフォントサイズをリサイズする。これらのリサイズはリソースファイルに記述することでも対応可能であるが、リソースファイルで記述できるものはその範囲が極めて限定される。さらにディスプレイのサイズに応じて Widget の配置も変更する。

次にメモリ容量とバッテリ残量に依存した振舞いとして文字入出力バッファの伸縮を行う。入出力バッファのサイズを大きく取れば、ディスクアクセスの回数を減少し、効率的な入出力を実現するだけでなく、ディスクアクセスに伴う消費電力の節減にも有効である。しかし、メモリが少ない場合は多少ディスクアクセスの回数が増えても、入出力バッファのサイズを抑えて、メモリの消費を節減したい。ここではメモリの消費を優先している。

そしてバッテリ残量に依存した振舞いとしてファイルの自動セーブ回数の増減を行う。このエディタはファイルの自動セーブ機能を持ち、バッテリの残量が少ない時には自動セーブの回数を抑えてディスクアクセスを減らし、バッテリの消耗を回避する。

4.2 リソース・モニタ

常に環境情報を必要とし、その情報をもとに時々刻々と変化するようなアプリケーションとして、やはり LEAD と Tcl/Tk を用いてリソース・モニタを作成した。これは文字通りリソースの状態を表示するものである。具体的には現在接続されているネットワークの種類、使用可能なメモリ容量、ディスプレイサイズ、電源の種類とバッテリ残量を表示する。このプログラムは前章で説明した動作をする。つまり基本的にポーリングしてシステム情報を拾うのだが、スリープする時間を状況に応じて変えながらポーリングする。例えばバッテリ残量は刻一刻と変化するものであるが、電源の種類が固定電源であった場合、バッテリ残量をさほど気にする必要はなくなる。この時にはスリープ時間を長くする。逆にバッテリ駆動時にはバッテリ残量は重要であり、スリープ時間を短くして正確な値を取得するようにする。また処理能力の高い CPU を持つマシンであれば、多少無駄な CPU 時間を費やしてもよいが、処理能力の低い CPU であった場合は、なるべく無駄な CPU 時間の消費を回避するためにスリープ時間を長めにとる、といった振舞いをする。

4.3 評価

本研究であげた例題の評価には二つの難しさがある。一つは数値として表せないものの場合である。例えばウインドウのリサイズ、Widget の配置替え等について、その有効性を数値で示すことはできない。また物理的な容量に伴う問題の場合、その節減の有効性については動作環境によって結果が異なる可能性が大きい。このため例題として作成したアプリケーションの厳密な評価を行うことは困難である。しかしLEAD 自体の有効性を示すことは可能である。つまり、このようなシステム構成・環境に依存し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。それはエディタの作成にかかったコーディングの量を比較すれば一目瞭然である。

- **Tcl/Tk のコード** = 約 1200 ステップ
- **LEAD のコード** = 総称関数 :6 メソッド :14

Tcl/Tk の約 1200 ステップに対して LEAD の記述量はごくわずかである[5]。しかも LEAD のコードはその大半が Tcl/Tk のコードであるといって差し支えない[6]。これはリソース・モニタにも当てはまる。このようなごくわずかの記述量で既存のアプリケーションに対して、システム構成や環境に動的に対応できる機能を付加することが可能となる。

5 結論

5.1 まとめ

以上、本研究で提案した言語処理系 LEAD の有効性について以下にまとめる。

- システム構成・環境の動的変化に対応する部分をアプリケーション本体から分離 (モジュール化) して記述できる。
- システム構成・環境の動的変化に応じたアプリケーションの振舞いを選択する共通の機構をアプリケーションに対して提供する。
- 既存の言語からなるアプリケーションに、容易にシステム構成や環境に依存した振舞いを実現する機能を付加することができる。
- 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によって記述することができる。
- メソッド選択のポリシーをシステム構成・環境に応じて柔軟に変更かつカスタマイズすることができる。
- プログラムのリモート実行を行なうことでシステム構成・環境によらずアプリケーションの実行効率を維持することができる。

また今後の課題として以下の点があげられる。

- LEAD_t の完全実装
- リモート実行機能の完全実装
- RT-Mach 上への LEAD の実装 (移植)
- アプリケーションの有効性の検証

本稿で述べた LEAD の機能のうち、LEAD_t 及びリモート実行は完全な実装を行っておらず、単純なケースのみサポートしている。また LEAD は RT-Mach [7] 上に実装 (移植) を予定している。RT-Mach は従来の UNIX などと比較してシステム構成の動的な変化に適した OS である。こうした RT-Mach 上で LEAD を用いたアプリケーションの有効性を検証することも今後の課題である。

[5] LEAD のコーディング量をステップで算出していないのは、見易すさの為の改行を行っているからである。

[6] 実行ソースコードの部分が最もコーディング量が多いため。

参考文献

[1] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, John H. Hartman. Scout: A Communications-Oriented Operating System, Department of Computer Science The University of Arizona Tucson, AZ 85721.

[2] 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

[3] Chris Maeda, Brian N. Bershad. Service wishout Servers, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

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

[5] 保木本 晃弘, 中島 達夫. オブジェクト指向に基づくモーバイルコンピューティングに適したカスタマイズ可能なＯＳ構築のための枠組, In Proceeding of the 10th Workshop of Object-Oriented Computing, 1994.

[6] 保木本 晃弘, 中島 達夫, 渡部 卓雄. 移動計算機環境に適した拡張可能なオペレーティングシステムの構成技術, 第 11 回ソフトウエア科学会全国大会, 1994.

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

[8] 栗原 邦彰, 中島 達夫. サービスプロキシを用いた移動計算機環境の構築法, 第 11 回ソフトウエア科学会全国大会, 1994.

[9] 宮田重明, 芳賀敏彦共著. Tcl/Tk プログラミング入門, オーム社, 1995.

[10] 中島 達夫, 渡部 卓雄, 保木本 晃弘. 移動計算機環境に適した柔軟なソフトウエアの構築法, 第 49 回情報処理学会全国大会, 1994.

[11] Pattie Maes. Computational Reflection (Ph. D. Thesis), technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[12] 寺岡文男. VIP:ホスト移動透過性を提供するプロトコル, 日本ソフトウエア科学会 コンピュータソフトウエア, Vol.10, No.4, 1993 年 7 月.

[13] 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.