ICOT Technical Report: TR-353

TR-353

オブジェクト指向言語を使用したOSの開発例

近山　隆

March, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

オブジェクト指向言語を使用したOSの開発例

近山隆
ICOT

概要

ICOT で開発した逐次型推論マシン PSI 上のオペレーティングシステム SIMPOS を，論理型言語にオブジェクト指向機能を融合した言語 ESP で記述した経験について述べる．

OS のような大規模ソフトウェアの開発に当たっては，ESP の持つオブジェクト指向機能が有効に働いた．設計段階では，オブジェクト指向機能の提供するモジュール間インタフェースの柔軟性の高さのため，各モジュールの独立性を高く作ってもユーザに柔軟なインタフェースを提供でき，提供機能を直交性よく整理できた．開発途上では，モジュール独立性の高さゆえに，漸次的にシステムを拡張して行くことが可能であった．効率面では，モジュール間の独立性を保ちながらコードを共用することが容易であることの利益が大きかった．

オブジェクト指向機能の導入により実行速度面での若干のオーバヘッドが生ずることはやむをえない．しかし，オブジェクト指向に記述することによってプログラムの改訂が容易になり，性能向上のためのアルゴリズムレベルからの大幅改訂が可能になった．この利益は，オブジェクト指向機能実現のための定数倍の実行時オーバヘッドとは比較にならない利益をもたらした．

1 はじめに

ICOT で開発した逐次型推論マシン PSI[6] 上のオペレーティングシステム SIMPOS[4] を，論理型言語にオブジェクト指向機能を融合した言語 ESP[1] で記述した経験と，その際に ESP のオブジェクト指向機能がどのように有効に機能したかについて述べる．

PSI は論理型言語のための良好なプログラミング環境を提供するため ICOT で新たに開発した，論理型言語のためのワークステーションである．PSI は論理型言語の専用マシンであるため，従来の汎用計算機の OS をそのまま移植することは不可能であった．論理型と手続き型では記憶領域の用い方などに根本的な相違があり，手続き型の考え方で作られた OS では，論理型言語で記述したアプリケーションプログラムとのインタフェースがぎこちなくなる恐れが強かった．そこで，PSI の OS である SIMPOS は，全く新規に設計 / 開発することとした．

最も普及している論理型プログラミング言語は Prolog であるが，Prolog にはプログラムを構造的に分割して行くモジュール化機能が欠けており，そのままでは OS のような巨大なシステムの記述には機能的に不十分であった．そこで，論理型言語とオブジェクト指向言語を融合した新言語 ESP を設計し，これを記述に用いることとした．

SIMPOS をまったくゼロから短時日で開発でき，その後の改良のための大幅な改訂作業もスムースに行えたことには，多重継承機能などのオブジェクト指向機能による，モジュール間インタフェースの簡素化やコードの共用化による利益が大きかった．また，性能向上のための大幅な改訂が容易になったことは，オブジェクト指向機能実現のための実行時のオーバヘッドを補って余りあるものであった．

2 PSI と SIMPOS

ICOT 研究所が発足した 1982 年当時は，論理型言語の実用的な処理系は数少なく，いずれも速度 / メモリ容量 / プログラム開発サポートのいずれかの面で不十分であった．プロジェクトの最終目標のひとつは高度並列推論システムの構築にあるのだが，この時点ではとりあえず実用になる論理型言語ソフトウェアの快適な開発環境を用意することが急務であった．そこで，ICOT では第五世代コンピュータシステムプロジェクトの最初の仕事のひとつとして，逐次型推論マシン PSI の開発に着手した．

PSI は，Prolog の基本機能にさまざまな拡張を施した機械語 KL0 を直接マイクロコードで実行する，論理型高水準言語専用マシンである．実用実行速度は開発当時の最高速の Prolog 処理系であった Dec10 Prolog と同程度以上の約 30KLIPS (一秒間に 3 万回の推論を行う速度) で，最大 80MB という大容量のメモリを備えている．[1] 入出力機器としては，磁気ディスク，LAN インタフェース，高解像度ビットマップディスプレイ，キーボード，マウスなど，マンマシンインタフェースの実現が可能なだけのものを備えている．

論理型言語の専用マシンという特異性のため，PSI の上に従来の汎用機上のオペレーティングシステムを移植することは困難であった．たとえ不可能ではなくても，従来の手続き型言語を基本としたシステムのために設計されたオペレーティングシステムでは，論理型言語によるアプリケーションプログラムとの整合性に無

[1] その後の Prolog 処理系の実装技術の進歩を取り入れた PSI の後継機種 PSI-II は，最大 400 KLIPS，実用速度で約 100KLIPS 以上を実現している[3]．

理が生じる恐れが多分にある．たとえば，論理型言語にはメモリの内容を書き換えると言う概念はないが，手続き型の OS ではこれをユーザインタフェースの基本方式とするのが普通である．従来の汎用機上の論理型言語システムでは，言語処理系がこの不整合をいちいち吸収している．しかし，この方式では OS の提供する機能をそのままの形でアプリケーションプログラムに提供することはできず，OS の機能が拡張されるごとに言語処理系も拡張しないとならないなど，不便が多い．そこで，新たに論理型言語との整合性のよい専用の OS, SIMPOS を開発する方針をとった．

3 ESP 言語

SIMPOS の記述には論理型言語にオブジェクト指向機能を融合した言語 ESP を新たに設計し用いることとした．本節では ESP の設計の目的と，その機能概要を述べる．

3.1 目的

PSI に接続する各種の入出力機器を有効に活用し，個人使用向けのワークステーションとして高度のマンマシンインタフェースを提供するためには，SIMPOS はかなり大規模なシステムにならざるを得ない．また，高速の実行機構と大容量メモリを持つ論理型言語専用マシン PSI の上では，アプリケーションプログラムも制約の多い従来の Prolog 処理系の上のものに比べてはるかに大規模なものを想定しなくてはならない．

こうした大規模プログラムの作成に，平板なプログラム構造しか持たない通常の Prolog 言語では不適当であることは明白であった．当時すでに Prolog に高度のモジュール構造を導入しようとする試みはいくつかあったが，いずれも機能が不十分であったり，実現上の効率に関する配慮が欠けていたりしており，適当なものがなかった．そこで，我々は SIMPOS および PSI 上のアプリケーションプログラムのの記述言語として，Prolog よりさらに高水準の言語 ESP を新たに設計し，これを用いることとした．

3.2 ESP のオブジェクト指向機能の考え方

ESP は，ミクロには論理型言語であるが，マクロにはオブジェクト指向言語である．情報受け渡しはユニフィケーションによること，実行制御がバックトラッキングによることについては，ESP は Prolog と同様である．一方，多数のプログラムモジュールをひとつのプログラムにまとめていくやりかたは，Flavors[5] などに似てオブジェクト指向になっている．

論理型言語のプログラムの実行は，与えられた公理 (= プログラム) に基づく定理証明の過程と見ることができる．ESP においては，ひとつひとつのオブジェクトは論理型言語における個別の公理系に対応する．ESP ではオブジェクトにメッセージを送ることは，そのオブジェクトの表現する公理系で定理 (= メッセージ) を証明 (実行) することにあたる．同じ定理が複数の公理系で成り立っていたとしても，公理系が異なれば証明の仕方が異なるのは当然である．これは，同じメッセージでもオブジェクトが異なれば振る舞いが異なることに対応する．

ESP の述語 (手続き言語の手続きや関数にあたる) には二種類がある．メソッドと呼ばれるものは，実行時に第一引数に与えられるオブジェクトによって呼び出し先が変わるようになっており，これが上述のメッセージパッシング機能を実現する．いまひとつは局所述語と呼ばれ，静的に呼び先が決まるようなものであるが，そのスコープはひとつのクラス定義 (後述) 内に限定される．メソッドをモジュール間インタフェースの記述に，局所述語をメソッドの詳細の記述に用いるのが普通である．

3.3 オブジェクトの状態

オブジェクトの状態が変化すると，同じメッセージに対する反応が変わる．これは，論理型言語として考えると，公理系の変更にあたる．ESP のオブジェクトは，その状態を保持するために，スロット と呼ばれる，ほぼ Smalltalk[2] のインスタンス変数にあたる記録場所を持っている．スロットの保持する値は任意に参照・更新ができる．

多くの Prolog では，公理系の変更は assert/retract と呼ばれる機能を用いて任意に行うことができるようになっている．ESP のスロットの機構は，一般的な公理の変更に比べて，非常に限定されたものである．このような限定された変更のみを許す仕様を採用した理由のひとつは，プログラムの実行時の書き替えにあたるような任意の変更を許すと，コンパイラによる静的な最適化がほとんど不可能になり，効率的な実装が困難になることにある．それ以上に問題なのは，実行中に任意に書き替わるようなプログラムは，その意味を読み取ることが大変難しいことである．プログラムが読みにくくなると，システムの保守性を大きく損なうことになる．

3.4 クラスとインスタンス

スロットの値だけが異なり，他の公理が同じであるようなオブジェクトは，ひとつのクラスのインスタンスとして定義する．こうすることによってインスタンス間でソースコード / オブジェクトコードの共用が容易になる．

クラスごとにひとつ，クラスオブジェクトと呼ばれる特別のオブジェクトがある．クラスオブジェクトは新たなオブジェクトの生成など，インスタンスオブジェクトの管理を行う目的で用いるのが普通である．ESP では通常のインスタンスオブジェクトは無名であり，名前でアクセスすることはできない．クラスオブジェク

トに限りクラス名を用いて参照することができる．ESP には大域的な変数が存在しないので，クラスオブジェクトは大域的な共用データの置場として用いることもある．

3.5 継承

クラスは親クラスをいくつでも持つことができる．オブジェクトの表わす公理系は，そのオブジェクトが属するクラス自身に記述された公理群と，親クラスに記述された公理群の合併集合になる．これが ESP の継承機構の論理型プログラム言語としての解釈である．

Prolog のような逐次型の論理型プログラム言語には，手続きの呼び出し (定理の証明) にあたって適用できる公理が複数ある場合，当面ひとつを選択して実行を続行し，後にそれで都合が悪くなると元に戻って別の選択肢を選ぶという，バックトラッキング機能がある．ESP の継承機構は，このバックトラッキング機能とあいまって，意味ネットワークの IS-A 関係を実現するものである．たとえば，親クラスが『動物』を表わすもので動物に関して適用できるような公理を持っていたとすると，『乳類』を表わすクラスを『動物』クラスの子クラスにしておけば，動物に対して適用できる公理はすべて乳類についてもあてはまる，ということを簡単に表現できる．公理の試行順は，子供クラスから親クラスへの順になっている．[2]

3.6 非単調機能

ESP の継承機構では，あるクラスの子クラスはすべて親クラスと同じ公理を合わせ持つことになる．親クラスにも子クラスにも同じ述語に対する公理がある場合は，どちらも適用可能になる．したがって，ある定理が親クラスの表わす公理系のもとで成立するならば，同じ定理はその親クラスのどの子クラスが表わす公理系でも成立することになる．こういう意味で，ESP の継承機構は原則として単調である．

これに比べて，Smalltalk などでは子クラスに親クラスと同じプロトコルのメソッドが定義されていると，親クラスでの定義は隠されてしまう．これを，オーバライディングという．ESP の単調な継承機構は，論理型プログラムの特徴であるバックトラッキング機能を生かした機構であると言える．

しかしながら，単調な継承機構だけでは，継承構成の一番上位に置くクラスは，すべての子クラスの機能を想定して定義しなければならない．このため，プログラム開発の前にあらかじめ完全な設計が必要になる，という問題点がある．たとえばクラス“鳥”についていったん“鳥は飛ぶ』という公理を与えてしまうと，『鳥』の子クラスとして『ペンギン』を定義し『ペンギンは飛ばない』などということができなくなる．

[2]多重継承の場合の試行順はやや複雑である．詳しくは [1] を参照されたい．

Prolog における唯一の非単調な機能は，カットと呼ばれる公理の選択を制限する機能である．ESP でも同じカットの機構を用いて，Smalltalk のようなオーバライディング機能を実現することができるようになっている．

ESP のもうひとつの非単調機能は，Flavors と同様の，デモンコンビネーション機構である．通常のメソッド定義は選択肢を追加する OR の働きをするが，デモンメソッドは公理の適用条件を追加する AND の働きをする．たとえば，ある大きさ以下にはできないウインドウを作ろうと思ったら，指定が制限内におさまっているかどうかを確かめるコードを，大きさを指定するメソッドの前デモンとして記述して置く．こうすると，実際に大きさを設定するコードの呼出しの前にこのデモンが AND されて自動的に呼ばれ，チェックにかかれば実際の設定にまでは到達しない．

このように，デモンコンビネーションは複雑な実行制御の実現を容易にする働きがあり，実用上のメリットは大きい．しかし，本来の論理型プログラムの枠からは，はみだしたものである．

4 SIMPOS の開発における ESP の役割

SIMPOS の記述には，ESP のさまざまな機能を活用した．ESP が論理型言語であるがための利益も非常に大きかったのであるが，ここではオブジェクト指向機能を中心に，どの機能がどのように有効であったかを個別に述べる．

4.1 仕様と実現の近さ

SIMPOS の具体的な開発に着手する前に，PSI の OS の要求仕様を設計した．この設計はオブジェクト指向におこなったが，これにより自然な設計が容易になった．たとえば，必要な機能をできるかぎり統一化していく上で，継承の概念はたいへん有用であった．

記述言語である ESP も継承機構などのオブジェクト指向機能を設けたため，要求設計を直接的に実装設計に写像することができた．このように仕様記述とその実現の間の距離が小さいことには，以下のような利点があった．

プログラム記述の容易さ：仕様とプログラムの記述言語の間にギャップがほとんどないので，仕様をかなり直接的に書き下すだけで実行可能なプログラムが得られる．

プログラムの読みやすさ：仕様の上での概念がプログラム上の概念とほとんど一対一に対応するので，必ずしも説明文書が完備していないプログラムを比較的容易に読解できる．ことにシステム開発途上には説明文書を完備することが困難なので，プログラム自身が説明文書として役立つことは非常に有用である．

仕様の問題点の発見の容易さ: プログラムの記述がどうしても不自然になる場合，プログラムの記述法ではなく，仕様の側に問題があることが少なくない．ESP で記述していると，仕様の問題点が忠実にプログラム記述の不自然さに反映するので，仕様上の問題点を発見しやすい．

4.2 動的なメソッド探索機構

ESP のメソッド呼び出し機構では，呼び出される相手を実行時の引数によって決めることができる．この機構には以下のような利点があった．

モジュールの独立性の高さ: 各モジュール(クラス)をほとんど独立に設計することができる．あるメソッドを呼び出す部分ではそのメソッドが『何を』するものかだけに着目すればよく，それを『どう』実現するかを気にしないで済む．これは通常の手続き指向のモジュール化方式でもいえることではある．しかし，手続き指向の場合は呼び出し先があらかじめひとつに定まっているために，その特定の呼び出し先が『どう』作られているかを意識してしまいやすい．オブジェクト指向言語の場合は，呼び出し先は実行時まで決まらないので，『何を』するものか以外は意識しにくいようになっている．

モジュール間インタフェースの柔軟性: ESP では，呼び出しの引数にオブジェクトを渡し，渡したオブジェクトのメソッドを呼ばれたコードの中から呼ぶ，といった記述方法も可能である．こうすると，引数として呼び出す側のモジュールで準備するのオブジェクトを渡して，呼び出されたモジュールの中での動きを高度にパラメタ化することができる．手続き型言語の手続き引数との違いは，手続きひとつだけではなく，一群の手続きとそこで用いるデータを合わせたもの全体をパラメタとすることができる点にある．これはセキュリティの面では危険なインタフェース方式であるが，非常に柔軟なインタフェースが実現できる利点がある．SIMPOS では，OS の異なるレイヤー間のインタフェースにこの方式を多用している．

システム拡張 / 修正の容易さ: 新しいモジュールをシステムに追加したり，一部のモジュールを改訂したりすることが非常に容易である．呼び出される側のモジュールを新たにシステムに追加したり改訂したり場合，呼び出されるコードを決めるのは呼び出される側なので，改訂対象でない呼び出し側のコードについては，コンパイルし直す必要すらない．

プロセスごとの割り付け資源の管理は，動的なメソッド探索機能が有効に働いた局面のひとつである．プロセス単位に割り付け，プロセス終了時に解放したい資源は，ファイルのオープンもデータベース更新時のロックも，まったく同じやり方でプロセスオブジェクトに登録しておく(実際にこの登録作業を行うのは，アプリケーションプログラムではなく，ファイルサブシステムなどのコードである)．プロセスの終了時にファイルを閉じることも，ロックの解除も，プロセス管理プログラム中から登録されている資源に向こ『解放せよ』というメソッドを呼び出すことで実現できる．プロセス管理側は『何を』すなわち『解放』をするのだ，という抽象的なレベルで記述しており，『どう』解放するかの詳細は資源側の問題，と独立性が高く保たれているわけである．

実際，SIMPOS の機能拡張にともなって，プロセス終了時に自動的に解放したいような資源の種類は次々に増えて行った．しかも，ひとつひとつの資源ごとに『解放』の仕方は異なっている．このような機能拡張の際に，いちいちプロセス管理モジュールを改訂せずに済んだのは，上述の方式を取れたためである．

4.3 継承機構

継承の機構により，よく似た機能を追加する場合には既にあるものとの差異だけを記述するという，いわゆる『差分プログラミング』が可能になる．これには以下のような利点がある．

コードの共用: 『差分』以外のコードは共用できるので，全体のコードのサイズを小さく保てる．これは，ソースコード・オブジェクトコードの両方について言え，巨大化しがちなシステムをなんとかコントロールできる大きさに保つのに有用であった．

プログラムの読みやすさ: 他のクラスとの『差分』だけを記述すればよいので，クラス定義は小さくでき，そのクラス特有の機能だけを記述することになる．大きなシステム全体を理解する場合，個々のモジュールごとに理解して行けばよいので，プログラムの解読が容易になる．

システム拡張の容易さ: 新しい機能を追加する際に，純粋に従来の機能に追加する部分のみを記述すればよいので，コーディングが容易である．もっと重要なことは，既存のコードには手を入れずに済むので，拡張機能をデバッグする際に，デバッグ対象を新規追加部分に限定できることである．

提供機能の直交性: ESP に多重継承機能があるので，アプリケーションプログラムも SIMPOS の提供するクラスを必要に応じて組み合わせて使うことができる．多重継承機能なしに，色々な機能の組み合わせをすべて用意するのは，OS が組み合わせ的に巨大化してしまう結果となる．また，従来の多くの OS に見られるように，あらゆ

る機能を合わせ持つものをひとつだけ提供する方針をとると，使わない余分な機能の存在のために効率の低下を招きやすいし，ユーザインタフェースが複雑化しがちである．

インタフェースの柔軟性: アプリケーションプログラムで OS の提供するクラスの定義の一部だけを差し替えたようなクラスを作ることによって，OS 機能をカスタマイズすることができる．セキュリティの面では問題があるが，あらかじめカスタマイズできる部分を決めておいてスイッチで切り替える方式に比べて，はるかに柔軟な方式である．この機能があるので，OS はあらゆるアプリケーションを想定し，それに必要な機能すべてを網羅的に提供する必要がなくなる．

管理の容易さ: あるクラスの修正が必要になった場合，その修正は自動的にすべての子クラスに引き継がれる．これによって，プログラム開発途上のモジュール間インタフェース管理が非常に容易になる．継承機構ではなくコードをコピーするような方法を取っていたならば，小さな修正でも，その影響範囲を正確に把握して適切に対処することは容易ではないことが多い．

SIMPOS で多重継承機能が最も効果的に働いたのは，ウインドウシステムである．SIMPOS のウインドウシステムは，OS が提供する部品をユーザが自由に組み合わせて，必要な機能を持つウインドウを構成できるようになっている．このため，ウインドウシステムは個々の部品を用意するだけで済み，あらゆる組合わせを提供するのに比べてはるかにコンパクトに記述できた．

5 実行効率について

ESP のオブジェクト指向機能の実現には，当然若干の実行時オーバヘッドをともなった．このオーバヘッドとオブジェクト指向機能による利益のトレードオフについて論じる．

5.1 オブジェクト指向機能のオーバヘッド

第一引数のオブジェクトによって呼び出し先が動的に決定するメソッドの呼び出しには，実行時に呼び出し先の探索が必要になる．親クラスをひとつだけしか許さない単一継承機能のみしかなければ，スロットのオブジェクト内相対位置は静的に決定できる．しかし，多重継承を許す場合，スロット位置を静的に決める方式ではコードの共用が困難になる．PSI 上の ESP 処理系ではコードの共用化を優先し，スロットアクセスの実行時にスロット Id と相対位置の対応表を探索する方式をとっている．

PSI 上の ESP 処理系ではこうした実行時の探索は，マイクロプログラムでコンパイル時に作ったハッシュ表を探索する方式で実現している．継承関係はコンパイル時に解析してひとつの表にまとめておき，継承に関わる探索は不要にしているので，探索に要する手間はほぼ一定である．具体的には，メソッド呼び出しの場合，静的に呼び出し先が決定できる局所述語の呼び出しに比べて約 3 倍，スロットのアクセスも通常の配列のアクセス (マイクロコードによるインデクス範囲の検査を含む) の約 3 倍の時間がかかる．これは決して小さなオーバヘッドではない．

5.2 プログラム改訂の容易性と実行効率

プログラムの効率を決める最も重要な要素は使うアルゴリズムの適否にあり，低レベルの記述の巧拙や処理系の効率ではない，というのは周知の事実である．しかしながら，小規模のシステムにあっては，最適なアルゴリズムを選択するのは容易であり，低レベルの実行効率が性能を左右する場合が少なくない．

OS のような大規模なシステムにあっては，全体として最適なアルゴリズムが何であるかをあらかじめ完全に把握することは容易ではない．したがって，システムが動作を開始してから各種の計測を行い，どこが性能のネックになっているかを解析し，かなり大幅なプログラムの書き替えを行わないと，なかなか満足できる性能が実現できないのが普通である．

最適なアルゴリズムが何であるかは当然要求仕様に依存する．ある程度長い期間用いられるシステムでは，時間と共に要求仕様が変わってくることが珍しくない．要求仕様に表向き質的な変化がなくても，利用形態の変化によって量的に変化があり，それによって最適なアルゴリズムが異なってくることも非常に多い．そうした変化をすべてあらかじめ予測することは事実上不可能である．したがって，初期の要求仕様に沿って最適化したシステムであっても，要求の変化に対応してアルゴリズムレベルからの大幅な変更が可能でないと，最終的に提供できる効率は低いものになってしまう．

こうした要素を考慮に入れると，言語の工学的な意味での実際的な実行効率には，プログラムの大幅な改訂がどの程度容易であるかが大きく影響することがわかる．言語のレベルを下げて実行効率を稼ぐことによって得られる速度向上は，せいぜい一桁程度である．一方，アルゴリズムを改善することによって得られる速度向上は，その程度には止まらない．

5.3 SIMPOS 開発時の大幅改訂の例

1984 年 7 月に SIMPOS のウインドウシステムが初めて動作を開始した時点では，一文字をキーボードから入力してビットマップディスプレイ上のウインドウに表示するのに数秒かかるという状態であった．この時点では，記述言語の実行効率に問題があるのでは，という声もあった．

しかしその後，データ表現形式，使用アルゴリズム，プロセスごとの役割分担などを全面的に見直し，わずか数か月の間に二桁以上の速度向上に成功し (新たに導入した ESP のためのファームウェアサポートによる約 3 倍の速度向上を含む)，同年 11 月初めに開催された FGCS'84 国際学会までには実用に耐える速度を達成できた．このウインドウシステムは任意にネスティングできるオーバラッピングマルチウインドウ機能を提供するもので，この時点で既にソースコードで一万行を越える大規模なものであった．改良点も非常に多岐に渡り，プロセス管理やプロセス間通信方式など，ウインドウシステム以外多くのモジュールにも及んだ．

このような巨大で複雑なシステムに対するこれだけの規模の改訂を，ごく短期間に実現できたことは，開発担当者の能力と努力によるところが大きいのはもちろんである．しかし，ESP による記述のモジュール独立性の高さ，モジュール間インタフェースの簡潔さと柔軟性，そして開発管理の容易性がなければ，このような大幅改訂はほとんど不可能であったろうことも事実である．

6 おわりに

SIMPOS の開発を通じて，その記述に用いた言語 ESP のオブジェクト指向機能がどのような局面でどのように有効に働いたかについて述べた．従来の OS の開発に比して，SIMPOS の開発に要した時間と労力は比較的少なくて済んだ．その大きな要因となったのは，記述言語 ESP の高いレベルの記述能力にあった．

非常に高水準な言語 ESP をシステム記述全般に用いることは，決して小さくない実行効率上のオーバヘッドを伴う．しかし，高水準言語によってもたらされる記述の柔軟性によって，効率向上のための大規模なシステムの改訂が容易になる．その利点は高水準言語の実行オーバヘッドによる損失を補って余りあるものであった．

謝辞

本稿の内容は PSI, SIMPOS の開発に携わった ICOT 内外の数多くの方々の努力に基づいたものである．ここに深甚なる謝意を表したい．

参考文献

[1] T. Chikayama. Unique features of ESP. In *Proceedings of FGCS'84*, ICOT, 1984.

[2] A. Goldberg and D. Robson. *Smalltalk-80 the Language and its Implemenation*. Addison-Wesly, 1983.

[3] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine PSI-II. In *Proceedings of 4th Symposium on Logic Programming*, 1987.

[4] S. Takagi et al. Overall design of SIMPOS. In *Proceedings of the Second International Conference on Logic Programming*, Uppsala, 1984.

[5] D. Weinreb and D. Moon. *Lisp Machine Manual, 4th edition*. Symbolics, Inc., 1981.

[6] M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida. *The Design and Implementation of a Personal Sequential Inference Machine: PSI*. ICOT Technical Report TR-045, ICOT, 1984. Also in New Generation Computing, Vol.1 No.2, 1984.