オペレーティング・システム 27 － 2
（１９８５．５．１０）

ＳＩＭＰＯＳのプログラミング環境

高木　茂行，近山　隆，坂井　公，佐藤正俊
黒川利明，服部　隆，辻　順一郎
（財）新世代コンピュータ技術開発機構

１．始めに

ＳＩＭＰＯＳは、第５世代計算機プロジェクトの前期３年間の主要な成果目標の１つとして開発が行われた逐次型推論マシンＰＳＩのオペレーティング／プログラミング・システムとして開発された。ＰＳＩ及びＳＩＭＰＯＳは現在ＩＣＯＴ及び関連８社において、計２０台以上が実稼働しており、第５世代プロジェクト中期以降の開発ツールとして利用可能な状態となっている。本報告ではプログラミング環境としてのＳＩＭＰＯＳの設計方針、概要、その実現について述べる。

２．設計方針

ＰＳＩは第５世代計算機の研究開発のツールとなるスーパー・パーソナル・コンピュータ／ワークステーションという位置づけがなされている。従来のこの種の計算機としてはＬｉｓｐ系、Ｓｍａｌｌｔａｌｋ系、Ｐａｓｃａｌ系、マイクロ・コンピュータ系等の各種のものが、実験あるいは実用に供されているが、Ｐｒｏｌｏｇ用のものとしてはＰＳＩが世界初のものである。

第５世代計算機で用いるベース言語としてＰｒｏｌｏｇを用いることを前提とし、Ｐｒｏｌｏｇの効率良い実行と、システム開発用の環境を提供するという目的のために、望ましい計算機という方針でＰＳＩは開発された。

従来の汎用機のＴＳＳではなくワークステーションにした理由は、他のユーザのプログラムの実行に影響を受けない／与えないこと、１ユーザで全ｃｐｕ能力を得られること、大容量の実記憶を占有できること等の理由による。性能的には３０KLIPS（ＤＥＣ－１０　Ｐｒｏｌｏｇのコンパイル済コードと同程度）、１６ＭＷ（１Ｗ＝８ビット・タグ＋３２ビット・データ）という値を目標とした。

ハードウェアに関してはこの目標値は現在の技術で充分に達成可能との見通しであったが、ツールとしての利用を考えると、この上に「使い易い」ソフトウェアを積み上げねばならない。ＳＩＭＰＯＳはこのためのオペレーティング／プログラミング・システムとして設計された。

実際のＳＩＭＰＯＳ開発にかけることのできるマン・パワーは、外注プログラマを中心に３０人前後×２年間が限度であり、従来のようなシステム開発体制では、前期３年間の終了時点にツールとしての使用に耐えるだけのワークステーションを作り上げることは不可能に近いと考えられた。従って、設計・コーディング・デバッグ・改良のどのフェーズも大幅な期間短縮を行なう必要があった。また、改訂が容易であることが重要であると考えられた。

一方、「使い易い」ユーザ・インタフェースを提供するためには、従来のワークステーションで提供されている機能のかなりの部分をカバーすることが必要である。特にビットマップ・ディスプレイを活用したウィンドウ・システムの活用が「使い易さ」に大きく影響すると考えられる。さらに、第５世代機のベース言語がＰｒｏｌｏｇである以上、ＳＩＭＰＯＳもＰｒｏｌｏｇで記述することを原則としたい。

以上の点から、Ｐｒｏｌｏｇにオブジェクト・オリエンテッドな機能を付加した言語ＥＳＰを設計し、それを用いて全システムを記述することとした。もちろんＰＳＩのハードウェア・ファームウェアもＰｒｏｌｏｇの実行に適した構成・機能・速度を持つように設計された。

これによってＰＳＩ－ＳＩＭＰＯＳはハードウェアからソフトウェアまでの全レベルに対してＰｒｏｌｏｇの実行メカニズムを基本とするという設計方針によって開発されることとなった。さらにＳＩＭＰＯＳは、小人数・短期間の開発を実行するため、オブジェクト・オリエンテッドな機能を利用したモジュールの利用を行ない、全システムにそれを徹底することによって工数を減らすという方針とした。

以上の点をまとめると次のようになる。

１）　ＰＳＩ－ＳＩＭＰＯＳはハードウェアからソフトウェアまでの全レベルに対して、Ｐｒｏｌｏｇの実行メカニズムを基本として設計・開発する。

２）　ユーザ・インタフェースの向上のために、ウィンドウ・システムを全サブシステムのユーザ・インタフェースに利用する。

３）　オブジェクト・オリエンテッドな機構を全ソフトウェアで利用し、工数の削減を行なう。

3．SIMPOS概要

SIMPOSは大別してOS部とPS部の2つに分類される。OS部は従来の汎用機でも同様のことが行なわれている部分で、メモリ管理等を行なうカーネル、プロセス管理等を行なうスーパーバイザ、周辺機器とのインタフェースとなるデバイス・ドライバ、ファイルを扱うファイル・サブシステム、ユーザ・インタフェースの中心となるウィンドウ・サブシステム、及び他のPSIとの接続を中心とした外部インタフェースのネットワーク・サブシステムから成る。

オブジェクト・オリエンテッドな考え方を全面的に適用しているため、個々のファイルやウィンドウは全てオブジェクトとしてとらえられる。操作はオブジェクトに対するメソッド呼出しの形で指定され、環境状態は引数で渡されるオブジェクトと、操作をほどこされるオブジェクトの内部状態を保持するスロットに保持されている。一方で、実行メカニズムはPrologであるから、一般の引数データはバックトラックによって結合が切られ、呼出し時に未定義だった変数は未定義の状態に戻される。オブジェクトやスロットは、バックトラックしてももとの値（状態）に戻ることはなく、副作用の対象として考えられる。従来の計算機における代入による値の設定はユニフィケーションで行なわれるが、これはバックトラックによってもとの値に戻ってしまうので、値の保存はスロットに副作用で設定することで行なう。

3．1　OS部

3．1．1　カーネル、デバイス・ハンドラ

OS部分のうち、特にハードウェアに近い部分であるカーネル、デバイス・ハンドラは、機能的には汎用機で行なわれるものと同様である。従ってバックトラックを必要とする処理はほとんど無く、ほとんどの述語が入力データを受け取って、何らかの動作を行なう手順をtail recursiveに記述してある。これは汎用機ではループによって次々に処理を行なうのと同じである。

メモリ管理の動作も汎用機で行なわれているのと同様のページ・マップによる管理を行なう。一方、ユーザ・プログラムでは一定のメモリ領域を扱う場合にはプールという概念を用いる。すなわち、プールのインスタンスを作ってそれに対して操作を行なう。従来はユーザ・プログラムの側で作られていた hash-bucketや linked-list等もクラス定義として提供されており、ユーザはプログラムを容易に作成できるようになった。

3．1．2　スーパーバイザ

スーパーバイザも、従来の汎用機とほぼ同様であるが、オブジェクト・オリエンテッドな機能を利用したプロセス間通信としてストリームの概念を取り入れた。プロセス間通信が行なわれる際には、当事者のプロセス間にストリームの経路を設けておき、それに対してそれぞれのプロセスはポートを持つものとする。ポートに対してメッセージとなるオブジェクトを送受信することによって、ポートからのデータ待ちの形でプロセスの待ちを実現する。これはメカニズム的には特に新しいものではないが、オブジェクト・オリエンテッドな方法によって実現を行ない、充分な機能と速度を実現し得た点に特徴がある。

3．1．3　ファイル

ファイル・サブシステムは、最近のマイクロ・コンピュータや小型コンピュータで多く用いられているようなディレクトリによる管理を行なうものとして設計した。

ファイルはそれ自体がオブジェクトとして扱われる。ユーザ・プログラムからはファイル・オブジェクトのインスタンスに対して入出力のメッセージを送る（メソッドを呼び出す）ことによって実際の入出力が行なわれる。

ファイルはワールドと呼ぶディレクトリに格納し、これを経由してパス名でアクセスされる。逐次的な読出／書込の処理には、ファイルに対してタップというオブジェクトを付加し、それを用いるのが一般的である。タップは主記憶中のメモリ領域を扱うプール・オブジェクトにも同じインタフェースで用いることができるので、ファイルかメモリかを意識しないで処理を書くことができる。

3．1．4　ウィンドウ

ウィンドウはSIMPOSのユーザ・インタフェースの中核をなすものである。ウィンドウ・サブシステムはビット・マップ・ディスプレイ上の複数のウィンドウを管理し、キーボードやマウスの入力を適当なウィンドウからの入力とみなして、対応するプロセスに送る。マウスによるメニュー選択を中心としたユーザ・インタフェースはLispマシンをはじめとするほとんど全てのスーパー・パーソナル・コンピュータに採用されており、その使い勝手は一般に良いとされている。PSIでもこの機能を用いて各種のプログラムを作成しているが、一般ユーザに対しては使い勝手は良いようである。

オブジェクト・オリエンテッドな機能を用いてウィンドウ・システムをユーザに提供する場合、ウィンドウの持つべき各種の機能をそれぞれ別のクラスとして定義し、それ

らの組合せによって自分のためのウィンドウを作成するという方法をとることができる。新しい機能を追加する場合、その機能を持つ新しいクラスを定義し、それを利用するクラスを作ることで新機能の取込みができる。この点でオブジェクト・オリエンテッドな機能は非常に柔軟なプログラミングを可能にすると言える。従来のプログラミング言語では、新しく自分用のモジュールを作るためには、自分のモジュールから他のモジュールを呼び出すための入口点を全てコーディングする必要があったが、オブジェクト・オリエンテッドな機能では必要なモジュール（クラス）を継承すれば全て完了となる。

３．１．５　ネットワーク

他のＰＳＩをはじめとした外部とのインタフェースはネットワーク・サブシステムが行なう。ＳＩＭＰＯＳ同志の範囲内では、全てオブジェクト・オリエンテッドな考え方が用いられており、他のＰＳＩ上のオブジェクトをネットワーク経由で、あたかも自分のＰＳＩのオブジェクトであるかのように扱うリモート・オブジェクトという概念が用いられている。ユーザ・プログラムからは、自分のマシン上のオブジェクトであるか外部マシン上のリモート・オブジェクトであるかを意識する必要はない。例えばファイルの転送を行なうには、ファイル・オブジェクトを先方のプログラムに送って、それを受取側でディスクに書込めば良い。

一方、ＰＳＩ以外の機種との接続はＰＳＩ同志のようなわけにはいかないので、専用のプログラムが必要である。現在ＲＳ２３２Ｃを用いた端末エミュレータが用意してあり、それによってファイル転送が行なえる。また、今年度中にイーサネットを用いたファイル転送プログラムを作成する予定である。

メイルやリモート・ログインについては、現在検討中であり、実現までには時間がかかりそうである。

３．２　ＰＳ部

ＰＳ部はユーザとのインタフェースを扱う部分であり、この部分の使い勝手がシステム全体の使い勝手として評価されると言っても、おおげさではない。ユーザとのインタラクションによって仕事をする各プロセスをここではエキスパートと呼ぶ。ＰＳには各エキスパートを仲介したり、ユーザ毎にワールドを設定するコーディネータと、各エキスパートとしてライブラリアン、ファイル・マニピュレータ、ターミナル・エミュレータ、エディタ、デバッガ／インタプリタ等がある。

ユーザは自分専用のエキスパートを登録することができ、コーディネータから必要なエキスパートを作成したり消去したりできる。また、どのディレクトリを自分の主ディレクトリにするかも指定できるので、自分専用の作業環境を確保することができる。

３．２．１　コーディネータ

コーディネータはエキスパートの管理を行なう。ユーザ毎に使用するエキスパートがワールドに登録されているので、それを用いてメニュー選択によってエキスパートを作成、削除する。また、ウィンドウ等のマネージャ・プロセスの呼出しもここで行なう。ユーザはコーディネータを用いて自分の作業環境となるエキスパートを起動し、それを用いて実際の仕事を行なう。

個々のウィンドウへのキーボードやマウスからの入力を切り分けて送る処理もここで行なわれる。エキスパート毎にキー・コマンドやマウス・クリックの翻訳表を持ち、同じキーでも異なる意味にして処理することもこのレベルで行なわれる。

プロセス間のオブジェクトの受け渡しは、一過性のものはホワイトボードを用いて行なう。継続的なものについてはストリームを用いる。コーディネータにはホワイトボードに関する管理も行なわせることができる。通常は、あるエキスパートがホワイトボードに出力したオブジェクトは、別のエキスパートが入力するとホワイトボードから消去されるが、ユーザはコーディネータからホワイトボードを操作してそれに介入することができる。

３．２．２　ファイル・マニピュレータ

ファイル・マニピュレータは、汎用機ではファイル・ユーティリティとして知られている機能のユーザ・インタフェースである。ファイルのコピーやプリント、ディレクトリの掃除等は、マウスからのメニュー選択によって全てここから行なうことができる。

３．２．３　ターミナル・エミュレータ

ターミナル・エミュレータは、ＯＳ部で述べたように，他機種とＰＳＩをＲＳ２３２Ｃで接続し、端末として使うためのエキスパートである。現在このエミュレータにはホストとの間にファイル転送をする機能も有しており、ソース・ファイルの転送が可能である。

３．２．４　エディタ

エディタはトランスデューサと組合せて、一般的な構造エディタとして用いることを前提に設計している。しかし現在のところ文字モードのみが実現されている。

トランスデューサは、文字列テキストとＰＳＩのタームの内部表現との間の変換を行なう。エディタの内部では，構造エディタとしての処理を行なうための内部表現にさらに変換して編集処理を行なう。

構造モードの場合、スクリーン上の表示はホロフラスティングにより、ユーザに見やすい形にして表示が行なわれる。そのためのプリティ・プリンタもトランスデューサの一部である。

文字モードの場合には全てのテキストがホロフラスティングなしに表示される。

コマンドについてはこれまで研究されてきたＬｉｓｐ等の構造エディタと似たようなものになると考えている。しかし、構造エディタにはタイプライタ端末を想定したものが多く、スクリーンを利用したものは例があまりないので、表示やコマンド系について細かい部分にわたる検討を行なっており、完成にはまだ時間がかかると思われる。

３．２．５　デバッガ／インタプリタ

ユーザがＰＳＩでプログラム開発を行なうためのインタフェースとして、デバッガ／インタプリタが用意されている。従来の汎用機におけるインプリメンテーションでは、インタプリタが中心にあり、それにデバッグ機能が付加されているという形式のものが多かったが、ＳＩＭＰＯＳではインタプリタとデバッガを分離し、デバッガを中心にすえた。ユーザの入力したタームを述語呼出しとして扱い、解釈実行するのはインタプリタであるが、実行の制御やトレース等はデバッガが行なう。機能を分離することによって、両者のインタフェースを変更しない限り、それぞれを自由に改訂できるようになった。

インタプリタはコンパイル済コードも解釈実行コードも全く同様に実行できる。もちろん、トレースや実行制御が行えるのは解釈実行コードに限られる。デバッガ／インタプリタのトップ・レベルからは、ＳＩＭＰＯＳの全てのプログラム（クラス／インスタンス）のメソッドやＰＳＩの組込述語を呼出すことができる。その意味で、デバッガ／インタプリタはＬｉｓｐマシンにおけるリスナの働きを持っていると言える。この中からオブジェクトの内容を見たり、変更したりする手段としてインスペクタを開発中である。インスペクタはデバッガのサブルーチンとして呼出され、渡されたゴール・タームの要素である引数やその内容についての操作を行なう。

３．２．６　ライブラリとライブラリアン

ＳＩＭＰＯＳで扱う全てのクラスは、ライブラリの管理下におかれている。ライブラリは各クラスのソース、テンプレート、オブジェクトの管理を行なう。これのユーザ・インタフェースとしてライブラリアンがあり、ユーザはクラスの登録、削除、コンパイル等をライブラリアンのメニューで指示できる。

ライブラリはＳＩＭＰＯＳのクラス管理の中核であり、クラスの継承・参照の管理もここで行なう。個々のクラスには、ソース、テンプレート、オブジェクトという３種の形態が存在する。ソースはユーザがコーディングしたプログラム・テキスト、オブジェクトはクラスの継承・参照が全て解決され、実行可能な形式になった主記憶上のイメージである。テンプレートは１個のクラスのソースからわかる情報を保持するもので、そのクラスに関するスロット、メソッド、ローカル述語の情報を含む。メソッド等の述語情報の中には解釈実行コードとコンパイル済オブジェクト・コードの両方を持つことができる。オブジェクトを作成する際にはどちらを使うこともでき、それによって解釈実行型のオブジェクトとコンパイル済型オブジェクトを使い分けることができる。デバッグが完了したクラスはコンパイルして登録し、デバッグ中のクラスは解釈実行することによって効率良くデバッグを進めることができる。

クラスの登録を行なう場合、継承・参照しているクラスは自動的にロードされる。これによってユーザは他のクラスを自分でロードする必要がなくなり、自分の作成したクラスのみを扱えば良くなる。ここで登録したクラスを実際に動かすためには、オブジェクトを作成する必要が生じる。ライブラリはオブジェクトの作成に必要な情報の収集を行ない、それらをまとめあげて１個のクラス・オブジェクトを作成する。

作成したオブジェクトの実行においては、解釈実行コードとコンパイル済コードが互いに呼びあうことが必要となる。ライブラリはこのインタフェースを提供し、必要ならばインタプリタを呼出すようにコードをリンクする。これによってユーザ・プログラムをデバッグすることが可能になる。

さらに、ライブラリは各クラスの持つスロット、述語の名前の管理を行なう。各クラスで定義されるスロット名、述語名はクラス内で閉じており、別のクラスで同名のスロットや述語を作ることができる。これらを異なるものとして識別するため、名称はライブラリの識別ＩＤと呼ぶユニークなアトムに変換して用いる。クラス名やメソッド名はクラス間にまたがって使われる名前なので、この変換は行なわない。

ライブラリの持つこれらの機能によって、ＰＳＩのオブジェクト・オリエンテッドな実行が裏付けられており、継承をうまく使用することによってプログラムの開発効率を向上させることができる。

４．オブジェクト・オリエンテッド・プログラミング

ＳＩＭＰＯＳの記述言語であるＥＳＰはＰｒｏｌｏｇの実行メカニズムであるユニフィケーションとバックトラックをもとにし、その上に多重継承を許したオブジェクト指向機能を持っている。継承関係の解決はオブジェクトの作成時に静的に行なわれ、実行時には固定された参照先の述語を呼び出す。

４．１　利点

オブジェクト指向機能を採用したことにより、以下のような利点があった。

１）　モジュール分割が容易
２）　組合せによるモジュール利用が容易
３）　モジュール修正が容易
４）　機能を小さくした分だけコーディングが容易
５）　モジュールを使用する側からは使用が容易
６）　類似モジュールを作成することが容易

このうち１～５はモジュール化の問題であって、手続き型言語のモジュール化機能の発展したものと考えることができる。オブジェクト指向による多重継承の利用により、単機能のモジュールを作っても、それらをまとめて継承すれば、必要な機能のそろったモジュールを作ることが可能となった。従来の手続き型言語では、複数モジュールの機能を合わせ持つモジュールを作るためには単機能モジュールを呼出すための入口点を作るとか、コードをコピーするとかの余分のコードを必要とした。継承によってモジュールに機能が付加できると余分なコードは一切不要となり、モジュール化をより容易にすることができる。

６はプログラムの再利用にかかわる利点である。ほぼ同機能で一部が異なったり追加されたりしたモジュールは、共通部分はそのクラスを継承することで済ませてしまい、相違部分だけのコーディングで作成することができる。プログラマの立場から見ると、これは

１）　作り易さ（開発・蓄積）
２）　直し易さ（保守・改良）

に利するものと言うことができる。特にＳＩＭＰＯＳの開発のように、小人数でかつ経験の少ないプログラマによって全体のコーディングを行なう場合に有効であったと考えられる。

４．２　欠点

オブジェクト指向言語について一般に言われているように、ＥＳＰにおいても最大の問題は実行速度が遅い点である。また、多機能のモジュールにさらに自分のための処理を付加したモジュールを作る場合、継承されるモジュールからさらに継承されているモジュールまでは目が届かないために予期しない動きをしたり、親のクラス（継承されるクラス）で必要なメソッドを子クラス（継承するクラス）で再定義してしまったために無限ループに入ってしまうといった問題が発生する。

実行速度の遅さは、オブジェクト指向である限りある程度はやむを得ない面がある。従来の手続き型言語では、呼出す手続きやデータの格納場所のアドレスは既に確定しているが、オブジェクト指向の場合は、呼出すオブジェクト毎にそのメソッドやスロットに対して異なるアドレスを用いる必要があるので、実行時にそのアドレスを求める処理が必須だからである。

この種の実行を高速化するためには、ハードウェアまたはファームウェアによるサポートが必要である。ＰＳＩではメソッドの呼出しと、スロットのアクセスを高速化をするためのファームウェア・サポートを行なっており、メソッドの呼出しオーバーヘッドはローカル述語の呼出しの３割強（メソッド呼出しにかかる時間はローカル述語呼出しの場合と比べて１．３倍強）、スロット・アクセスは全てソフトウェアで行なった場合の１０倍程度まで高速化されている。

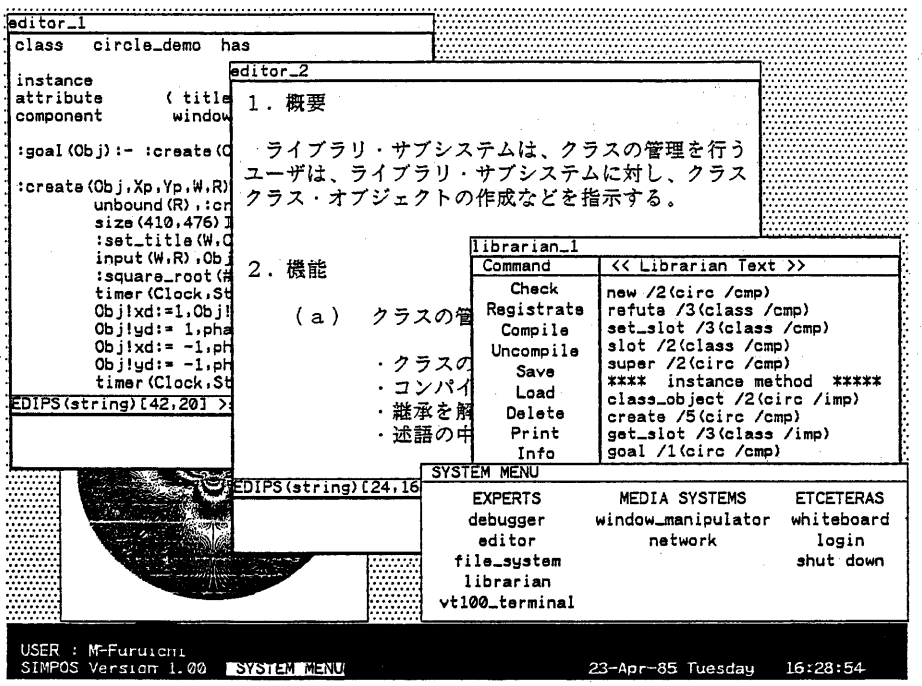
第２の問題は、継承によるソフトウェア蓄積と背反の関係にある。蓄積されたソフトウェア・モジュールは、その内部に持つスロットや、受付けるメソッドの全てを知って利用すればこの問題は生じないが、それではＯＳの全モジュールのことを知らなければならないということになりかねない。実際的な解決法としては、親クラスが定義しているメソッドの再定義（ｏｖｅｒｒｉｄｉｎｇ）をチェックして、警告を発する程度のことしかできそうにない。

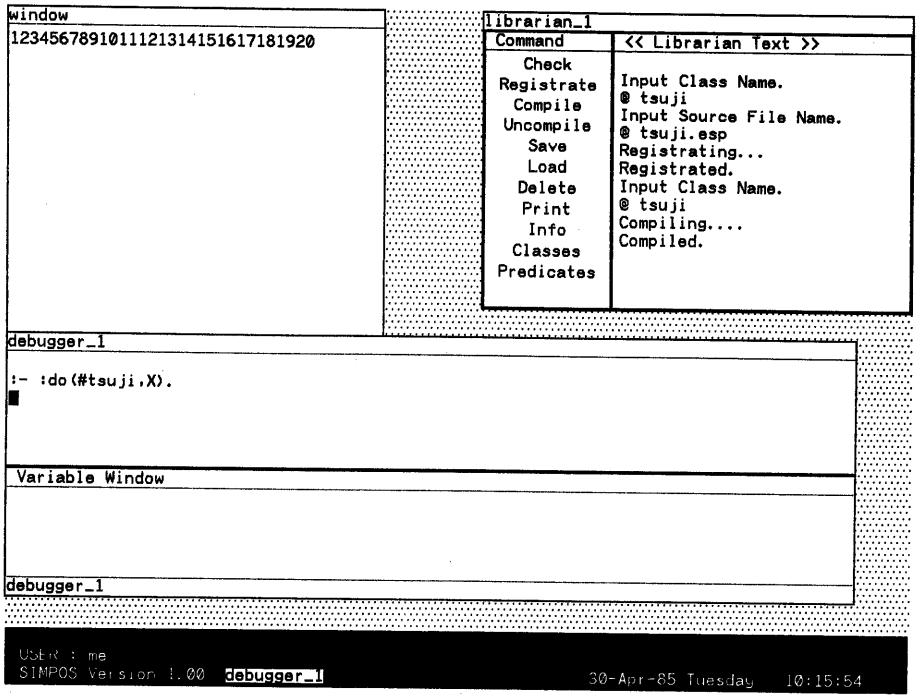
５．終りに

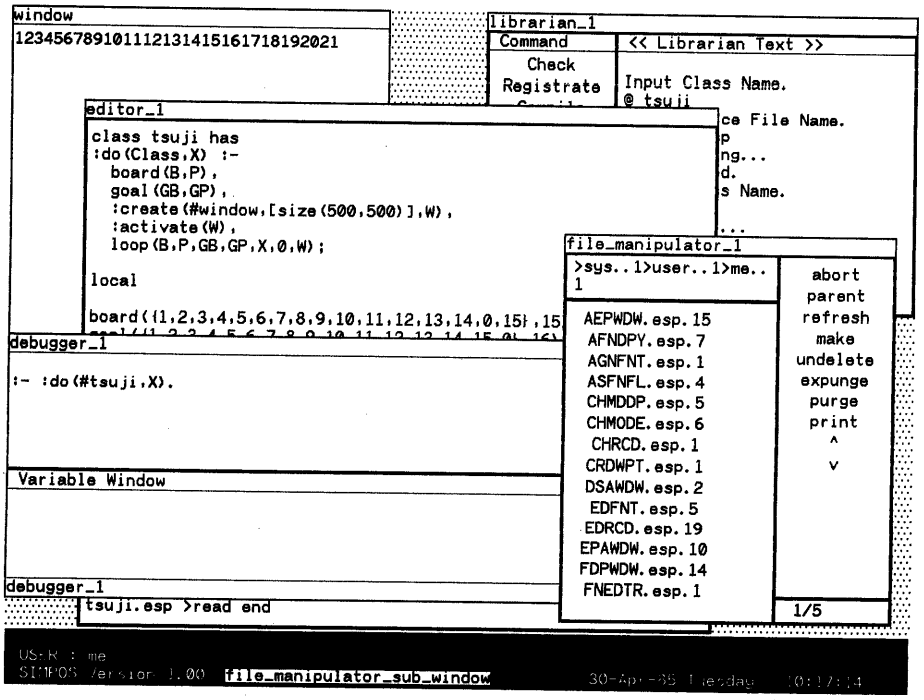
ＳＩＭＰＯＳの設計方針、概要及びその実現について述べた。Ｐｒｏｌｏｇの実行メカニズムとオブジェクト・オリエンテッド機能を合せた言語による、スーパー・パーソナル・コンピュータのＯＳとしてのＳＩＭＰＯＳが実用的なものとして働くことは示すことができたが、プログラミング環境としてのＳＩＭＰＯＳ（ＰＳ部分）は今後のアプリケーション構築によって評価されるものであり、今後も改良を加えていく予定である。

〈参考文献〉

服部他：SIMPOSのオペレーティング・システム
第29回情報処理学会全国大会予稿　4E-1～10(1984)
黒川他：SIMPOSのプログラミング・システム
第30回情報処理学会全国大会予稿　4E-1～11(1985)
Goldberg, Robson : Smalltalk-80, Addison-Wesley,
p.714 (1983)
Weinreb, Moon : Lisp Machine Manual,
Symbolics inc, P.532 (1981)
Bowen et.al, : DECsystem-10 PROLOG USER'S MANUAL,
Univ. of Edinburgh, (1983)




window
123456789101112131415161718192O
librarian_1
Command
<< Librarian Text >>
Check
Registrate
Compile
Uncompile
Save
Load
Delete
Print
Info
Classes
Predicates
Input Class Name.
@ tsuji
Input Source File Name.
@ tsuji.esp
Registrating...
Registrated.
Input Class Name.
@ tsuji
Compiling....
Compiled.
debugger_1
:- :do(#tsuji,X).
Variable Window
debugger_1
USER : me
SIMPOS Version 1.00 debugger_1
30-Apr-85 Tuesday 10:15:54


window
123456789101112131415161718192021
librarian_1
Command
<< Librarian Text >>
Check
Registrate
Input Class Name.
@ tsuji
editor_1
class tsuji has
:do(Class,X) :-
board(B,P),
goal(GB,GP),
:create(#window,[size(500,500)],W),
:activate(W),
loop(B,P,GB,GP,X,0,W);
local
board([1,2,3,4,5,6,7,8,9,10,11,12,13,14,0,15],15
file_manipulator_1
>sys..1>user..1>me..1
AEPWDW.esp.15
AFNDPY.esp.7
AGNFNT.esp.1
ASFNFL.esp.4
CHMDDP.esp.5
CHMODE.esp.6
CHRCD.esp.1
CRDWPT.esp.1
DSAWDW.esp.2
EDFNT.esp.5
EDRCD.esp.19
EPAWDW.esp.10
FDPWDW.esp.14
FNEDTR.esp.1
abort
parent
refresh
make
undelete
expunge
purge
print
^
v
1/5
debugger_1
:- :do(#tsuji,X).
Variable Window
debugger_1
tsuji.esp >read end
USER : me
SIMPOS Version 1.00 file_manipulator_sub_window
30-Apr-85 Tuesday 10:17:14