動的複合実行方式に基づく統合的プログラミング環境

筑波大学・電子情報工学　佐藤　豊　板野　肯三
Yutaka Sato　Kozo Itano

1．はじめに

プログラミングの方法論の有用性は十分認識していても、不完全な人間としてのプログラマがその教えを忠実に守ることは容易ではない。迷えるプログラマとしては、形式的な教えよりも、実際にその実践を励行し、誤りを監視してくれ、道を誤ってしまったときには速やかに迷路から救い出してくれる実践的なプログラミング環境が欲しい。すなわち、プログラミング環境は、プログラミングの各過程で、プログラマに要求されて個々のツールを提供するという受動的な形ではなく、行うべき仕事の流れを理解して先導するとともに、繁雑な仕事を裏方として引き受けてくれる能動的なものであって欲しい。

このような要求に答えるために、良いプログラミング環境は（1）統合性、（2）会話性、（3）高速性の3点を満たさなければならない。統合的な環境は、プログラミングの過程で頻繁に行われるツール間の移動と交信を高速化、さらには自動化し、一貫性のあるユーザ・インターフェースを与える。また、会話的な環境は、高水準で簡潔な指令に対して高速に的確な応答を、適切な表現形式で与え、人間の思考時間を短縮し、無為な待ち時間を削減する。さらに、プログラムの実行が高速に行える環境は、一定時間により多くのテストを尽くすことを可能にして誤りの早期発見を助けるとともに、デバッグ時には誤りの発生する現場により高速に到達することを可能にし、待ち時間を削減する。

このようなプログラミング環境を実現するためには、その基礎にあるプログラムの翻訳・実行系が、これらの要求に答えるものでなければならない。しかし、従来用いられてきた翻訳・実行系はいずれも、これらの要求を完全には満たしていない。

現在、ほとんどの”実用的”プログラムは、コンパイラをベースとする手続き型言語で書かれており、コンパイラをベースとするプログラミング環境で開発されている。一般にこのような環境では、オブジェクトコード・レベルで表現されたプログラムの実行の流れや状態の理解しにくさ、コンパイルに要する時間による応答性の低さなどのために、デバッグは容易でない。また、一つのプログラムに対してソースコードとオブジェクトコードという全く別の表現形式が併存し、異種の表現形式を扱うツール、例えばエディタとデバッガなどの間での交信が困難であることから、ツールの統合化が進めにくい。

一方、ソースコードの構造や意味を直接的に解釈し実行するインタプリタをベースとする環境では、プログラムの実行の流れや状態が、ソースコード・イメージで把握、変更できること、および実行に先立つコンパイルを要しないこと、などから会話性に優れ、デバッグが容易である［1］。また、プログラムの統一的な表現形式であるソースコードを中心として、各種のツールを統合することが容易であるため、INTERLISP［2］や CORNELL PROGRAM SYNTHSIZER［5］などでは、そのような統合的環境が実現されている。しかし、ソースコード・インタプリタでの実行が低速であること、および従来”実用的”言語に対するインタプリタがほとんど作られていないことから、その利用は特定の分野に限られてきた。

コンパイラをベースとした環境の問題点の解決法として、GANDALF［3］やDICE［4］では、インクリメンタル・コンパイルと呼ばれる方式を採用している。これは、構造化されたソースコードに基づいて微細な単位で部分コンパイルと部分的リンクを行うことにより、コンパイルを高速化するものである。また、構造化されたソースコードをツール間の交信のための共通表現として用いることによって、統合的環境を実現している。しかし、デバッグはやはりオブジェクトコードでの実行を基礎として行わなければならないことから、インタプリタ

をベースとする環境で得られるほどのデバッグの容易さは得られない。

一方、一般にインタプリタをベースとする環境では、実行の高速化のために、頻繁に参照されるプログラムの一部をコンパイルして実行することが行われているが、インタプリタの実行環境の中でオブジェクトが実行されるため、コンパイルによる高速化の効果が制限されている。

プログラミングにおける会話性を高めるには、ソースコードでの実行が、プログラムの実行を高速に行うにはオブジェクトコードでの実行が向いていることは、明らかである。そこで、この両者を組み合わせ、両者の利点を生かす動的複合実行（ＤＨＥ：Dynamic Hybrid Execution）と呼ぶ方式［６］を採用する。ＤＨＥ方式は、ソースコードでの実行が必要とされる部分はプログラム全体の一部分であり、かつプログラムの実行に従って動的に移り変わることに注目し、ソースコードで実行する部分を実行時に動的に切り換えることによって、必要な会話性を維持し、かつ実行の高速性を実現する。また、ソースコードを木構造化して表現するとともに、動的な切り換えを実現するためにソースコードとオブジェクトコードを常に対応づけて管理し、これを基礎にツールの統合化を実現する。

現在、ＤＨＥ方式に基づいた、言語Ｃを母体言語とする統合的プログラミング・システム、ＩＰＳ（Integrated Interactive Programming System ）を、ＶＡＸ１１上のＵＮＩＸを用いて開発している。以下、２章にその設計方針、３章でＩＰＳの全体像を述べる。そして、ＩＰＳの基礎となっているＤＨＥ方式について４章で、動的なリンキング方式について５章で述べ、最後に６章で簡単な例を用いて、ＩＰＳ上でのプログラミングの過程を説明する。

２．統合的なプログラミング環境の設計方針

手続き型言語による”実用的”なプログラムの開発を対象とし、計算限界なプログラムやリアルタイム・プログラムなどの開発にも利用できる環境の実現を目標とする。利用者としてはプログラミングの初心者も対象とするが、熟練者にも押しつけがましくなく、さらにプログラミング環境を自らの手で自分に合わせて変えて行こうとするプログラマの要求にも答えることができるような環境を目指す。

２．１　統合性

単なるツールの寄せ集めでなく、それらを一体化した統合的なプログラミング・システムとしてのプログラミング環境を実現する。各ツールは密接に結合されて互いに利用し合うとともに、情報と機能および状態を共有する。状態としては、作り出されたプログラムの実行状態だけでなく、それを作り出すプログラミングの過程の状態と流れを共有し、各ツールの起動を適切な時期に自動的に行う。そして、ユーザに対して一貫性のあるインターフェースを与えるために、ユーザ・インターフェースを一つに絞る。さらに、各ツールをユーザのプログラムと同一の空間に同一レベルのプログラムとして置き、ユーザの手によるプログラミング環境の変更を容易にする。また、これによって各ツールの状態の保存と回復を、ユーザプログラムの状態の保存、回復と同一の機能で実現する。

２．２　会話性

プログラミングには非定型で試行錯誤的な作業を伴うので、会話性は重要な要素である。そこで、会話を容易にするために、系統化された操作し易いコマンド体系を用意し、必要な情報を高速にディスプレイに分かり易く表示する。会話コマンドは、ユーザの手によって、ユーザの好みに合わせて、柔軟に変えられるようにする。また、会話の水準を高めるために、ソースコード実行方式を採用して、プログラムの作成、編集時の表現と、実行時の表現を一致させる。これにより、実際のプログラムの実行と、人間が仮想的に頭の中で行う実行のイメージが一致し、実行中のプログラムとの会話が全てソースレベルで行われるので、実行の制御と追跡、状態の観察が容易になる。

２．３　応答性

応答性を高めることによって、無駄な待ち時間を削減し、その時間に人間の頭の中にあるコンテクストが失われることを防ぐ。このために、システムはユーザの入力に対して常にできる限り高速に応答するように設計する。プログラムの変更が頻繁に繰り返される場合、変更に伴う翻訳と再リンキングに要する時間は応答性を著しく低下させる。この問題を解決するためには、変更と同時に実行開始が可能なソースコード実行と、変更を加えられた部分だけを対象する部分的再リンキング方式を採用する。

２．４　実行の高速性

テストの高速化を実現するために、ユーザの作成したプログラムをコンパイルして、高速に実行する方式も採用する。コンパイルはソースコードの変更に伴って自動的に行い、プログラマの思考時間を利用して”裏”で行う。計算限界なプログラムやリアルタイム・プログラムのデバッグ時には、会話性だけでなく実行の高速性が不可欠なので、以下で述べるような実行方式によって、この両者を両立させる。

２．５　実行方式

会話性と実行の高速性を両立するために、会話性に優れたソースコードでの実行と、高速なオブジェクトコードでの実行を組み合わせる。プログラムを部品の集合ととらえると、どのような開発方法をとるにしても、開発過程にあるプログラムに含まれる各々の部品の完成度は、一様でないのが普通である。このうち、ソースコードでの実行が必要とされるのは、誤りを含むまたはその可能性の高い部品であり、完成度の高い部品、たとえば組み込み関数やライブラリ・ルーチンをソースコードで実行することの意味はほとんどない。そこで、ソースコードでの実行を、それが必要とされている部品だけに適用することによって、必要な会話性を維持したまま、プログラム全体としての高速性を実現する。さらに、デバッグ時には、現在デバッグの対象として注目している部品だけがソースコードで実行されればよいので、デバッグの進展に従って、ソースコードで実行する部品を動的に切り換える。このように、ソースコードでの実行とオブジェクトコードでの実行を動的に切り換えながらプログラムを実行する動的複合実行方式（ＤＨＥ：Dynamic Hybrid Execution）［６］を採用する。ソースコードでの実行では、実行時エラー、例えば未定義な値への参照や、データ型の不整合のチェックを自動的に行う。

２．６　リンキング方式

一般的なプログラミング環境では、スコープはコンパイラによって、モジュール間の外部参照はリンケージ・エディタによって、プログラムの実行以前に解決される。このような静的リンキングの手法は、一度だけリンクしておけば、実行時に毎回リンクする必要が無いことから、完成したプログラムは向いている。しかし、頻繁な変更と部分的な実行が繰り返される開発中のプログラムの場合、静的なリンキングでは、実際には参照されない名前に対してもリンキングが必要であり、変更の影響を受けない部分も、再ローディングが必要になる。そこでこの問題を解決するために、動的なリンキング方式と、ブロック構造の動的な形成方式［６］を採用する。

３．統合的プログラミング・システム：ＩＰＳ

２章で述べた設計方針に基づき、会話性と高速性の両立を目標とする統合的なプログラミング・システムを開発しており、これをＩＰＳ（Integrated Interactive Programming System ）と呼んでいる。実現は、ＶＡＸ１１／７５０上のＵＮＩＸ／４．２ＢＳＤ［８］上で行っている。

３．１　システム構成

図１にＩＰＳのシステム構成を示す。システムの構成要素は、ユーザとの統一的なインターフェースであるユーザコマンド・インタプリタ（ＵＣＩ）、言語Ｃに対する木構造で表現されたソースコードを生成する構文指向エディタ（ＥＤＴ）とそれを実行するソースコード・インタプリタ（ＳＣＩ）、オブ

ジェクトコードを生成するコンパイラ（ＣＰＬ）とそれを実行するオブジェクトコード・インタプリタ（ＯＣＩ）、プログラムを構成する部品を動的に選択し、結合する動的リンキング・ローダ（ＤＬＬ）と、プログラムの潜在的構成要素である全ての部品を保持しその歴史を管理するバージョン管理システム（ＶＭＳ）、そしてこれらのサブシステム間の自動的な切り換えを行うＤＨＥカーネル（ＤＨＥＫ）から構成される。図でサブシステム間を結ぶ線は、サブシステム間での主な呼び出しと、データの受渡しの関係を示している。

３．２　プログラムの表現形式

ＩＰＳではプログラムの構造を、Ｃプログラムの個々の広域データや手続きを最小単位として認識し、これをプログラム・アトム（ＰＡ）と呼ぶ。ユーザは任意のＰＡの集合としてプログラム・モジュール（ＰＭ）を構成できる。ＩＰＳは、個々のＰＭをその名前とバージョン番号の組によって一意に識別し、これを単位としてバージョン管理を行う。プログラムはＰＡに関する参照関係を持つＰＭの集合として構成され、実行に伴ってロードされたＰＭをアクティブＰＭと呼び、それ以外のＰＭをパッシブＰＭと呼ぶ。

ＰＭは、ヘッダ部（ＰＭＨ）、ソースコード部（ＰＭＳ）、およびオブジェクトコード部（ＰＭＯ）から構成される。ＰＭＨは、そのＰＭの属性情報として、バージョン情報、参照回数やカバレッジなどの実行状況に関する情報、ＰＭＳやＰＭＯなどの配置やサイズなどを含むほか、そのＰＭで定義または参照されているＰＡを記述するＰＡディスクリプタのテーブルを含んでいる。ＰＭＳは木構造表現されたソースコードを含み、ＰＭＯは一次元的なＶＡＸ１１のマシンコード列、２進表現されたデータ、および再配置情報を含む。ＰＭＨ中のＰＡディスクリプタは、それが指すＰＡの名前やデータ型、配置などを記述する他、そのＰＡがソースコード、オブジェクトコードのいずれで実行されるべきかを指定する、”実行モード”を含む。

アクティブなＰＭ中のＰＭＳは、ＥＤＴによる編集によって書き換えられ、ＰＭＯはプログラムの実行に伴うデータへの書き込みによって書き換えられる。このように変更された任意の時点におけるＰＭは、それぞれ異なるＰＭのバージョンとして、保存、回復することができる。また、実行時のスタックやヒープの領域も、あらかじめ用意されたＰＭとして、同様にバージョン管理の対象になる。実行に伴って他のＰＭに結合されたＰＭを保存し、それを回復して実行を再開した場合には、保存時のＰＭ間の結合関係が再現される。

３．３　サブシステム

以下ではＩＰＳの構成要素である個々のサブシステムの機能と、サブシステム間の関連を述べる。これらのサブシステムはそれ自身ＰＭであり、ＶＭＳ中にあらかじめ用意されたＰＭである点を除けば、ユーザが普通に作り出すＰＭと同様に扱われ、ユーザプログラムから参照することもできる。

（１）　ＵＣＩ：ユーザコマンド・インタプリタ

ＵＣＩはプログラマとＩＰＳ間で会話を行うためのサブシステムであり、ユーザから受け取ったコマンドを実行する。UCIはマクロプロセッサであり、受け取ったコマンドを文字列としてのＣのソースコードにマクロ展開し、これをＥＤＴを呼び出して木構造で表現されたソースコードに変換し、ＳＣＩによって実行する。また、コマンドとして言語Ｃのソース・ステートメントを直接入力して実行することもできる。過去に入力されたコマンドの列は、ＰＭとして保存され、コンパイルされているので、その再実行は、オブジェクトモードで高速に行われる。ＵＣＩはＩＰＳの起動時に自動的に実行されるほか、キーボードからの割り込みや、実行中のユーザ・プログラムからの呼び出し、さらに他のサブシステム実行中にプログラマとの会話が必要とされるとき、随時起動される。プログラマはＵＣＩを介して他のサブシステムを起動したり、プログラムの実行状態を観察、変更することができる。

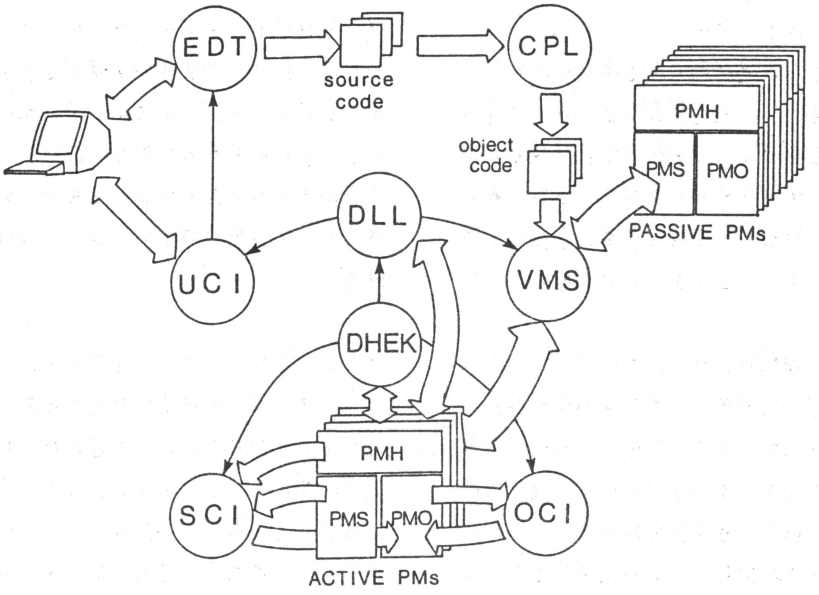

図１．ＩＰＳのシステム構成

（２）　ＥＤＴ：構造エディタ

ＥＤＴはＰＭ単位で言語Ｃのソースコードを編集する。ＥＤＴは構文指向エディタであり、一次元的な文字の列としてのソースコードを受け取り、これを抽象構文木形式の内部表現［１０］に変換する。また、この木構造表現を一次元的な文字列に変換して画面に表示する機能を持つ。ＥＤＴのコンテクスト、例えばカーソル位置などはＰＭＨ内に記憶され、ＰＭの編集終了後、そのＰＭが再び編集されるときに回復される。ＥＤＴは、ソースプログラムの生成、変更、表示が必要とされるときに、ＵＣＩやＳＣＩから起動される。ユーザはＥＤＴ中からＵＣＩを呼び出して、通常のコマンド入力状態に移行することができる。ＥＤＴによって変更が加えられたＰＭのオブジェクトコードは無効にされるとともに、自動的にコンパイルが起動される。

（３）　ＳＣＩ：ソースコード・インタプリタ

ＳＣＩは木構造表現されたソースコードをたどりながら、これを実行する。ＳＣＩは、一つの手続きの中の世界だけを解釈実行するインタプリタであり、ソースモードで手続きが呼ばれるごとに、ＳＣＩのインスタンスが一つ生成される。ＳＣＩは、ソースコードの情報に基づいて、未定義値の参照などの実行時エラーのチェックを行う。オブジェクトモードでの実行と等価性を保ち、整合させるために、２つの実行モード間で共有されるデータはオブジェクトコード・レベルに表現を統一され、ＰＭＯ中にとられる。ＳＣＩはそのデータの型などを記述するディスクリプタ［９］を介してこれにアクセスする。また、手続き呼び出しおよび戻り時の規約、すなわち引数や返値の渡し方をオブジェクトモードでの規約に合わせる。ＳＣＩは実行中に参照された手続きがソースモードであるとき、ＤＨＥＫから呼び出される。また、ＳＣＩは文法的に不完全なソースコードを実行することができるが、実行途中で不完全なソースコード部分（スタブ）に出会った時、ＥＤＴを呼び出してその完全な定義をユーザに求める。この際、ＳＣＩは、現在実行しているソースコード上の”プログラムカウンタ”にあたる木構造上のノード位置をＰＭＨ内に記憶しておき、これをＥＤＴが利用して、カーソル位置を決める。

（4）　CPL：コンパイラ

CPLはEDTから文字列として表現されたソースコードを受け取り、これを既存のコンパイラでコンパイルする。コンパイラは他のサブシステムと並列なプロセスとして実行され、EDTによって変更が加えられたソースコードが送られるのを待って受け取り、生成したオブジェクトコードを返す。

（5）　DLL：動的リンキングローダ

DLLは、実行中に参照されたPAの名前をPAの実体のアドレスに結合する。この結合時に、参照側の仮定するPAのデータ型やバージョンなどの属性と、被参照側の属性の整合性を検査する。このとき、参照されたPAの実行モードが未定義であるなら、それを決定する。DLLは実行中に参照されたPAがまだリンクされていないときに、DHEKから起動される。もし、対応するPAの実体がまだロードされていないときは、VMSを介してそのPAを含むPMを探してロードする。さらに、参照されたPAがVMSに知られていない（まだ作られていない）とき、UCIを起動してプログラマに制御を渡し、被参照PAの生成、あるいは参照側の綴り誤りの訂正などを行わせた後、再びロード・リンクを試みる。

（6）　VMS：バージョン管理システム

VMSはプログラムの潜在的構成要素であるPMの全てのバージョンを管理し、過去のバージョンの取り出しや消去、新たなバージョンの生成と保存を行う。変更されたPMのバージョン番号はVMSによって自動的に更新されるが、これを保存するか否かは、ユーザが指示する。VMSに含まれる機能としては、PA名を与えられて、それを含んでいるPMの一意名を返したり、PMの一意名を与えられてそれを保存または回復することなどがある。VMSは、動的なローディングのためにDLLから起動される。また、ユーザはUCI上またはプログラム中からVMSを直接的に呼び出し、明示的にバージョンの生成や保存などを起動する。

（7）OCI：オブジェクトコード・インタプリタ

OCIはVAX11のプロセッサおよびUNIXオペレーティング・システムそのものであり、オブジェクトコードを実行する。オブジェクト・モードで参照されたPAのコンパイルが未終了であったとき、OCIの実行はそのコンパイル終了まで待たされる。

（8）　DHEK：DHEカーネル

DHEKは、実行中に発生したPAへの参照を橋渡しする。DHEKは、参照されたPAの実行モードを、関連するPAディスクリプタ中のモード指定に基づいて決定し、参照されたPAがオブジェクトモードであるなら、そのオブジェクトコードを実行し、ソースモードであるなら、そのPAを記述するPAディスクリプタを引数としてSCIを呼び出す。PAディスクリプタ中の実行モードの指定には、実行モードを直接的な値としてだけでなく、実行モードを返す関数として指定することができる。このような指定に対しては、指定された関数を呼び出して実行モードを決定する。さらに、参照されたPAの実体がまだロードされていないときには、DLLを起動する。

IPSには、デバッグのための機能を提供する独立した”デバッガ”は存在せず、UCI、SCI、EDTが協同してデバッガの役割を果たす。

ユーザは、これらのサブシステムをプログラム中から呼び出すことによって、通常の環境では得られない機能を利用することができる。例えば実行中のユーザプログラムから、EDTに文字列を与えてプログラムを作り出し、実行することができる。また、UCIにユーザ定義のコマンドを与え、コマンド体系を好みに合わせて変えたり、ユーザプログラムのコマンド解析手続きとして用いることができる。さらに、もとから用意されているこれらのIPSのサブシステムを、プログラマが自由に変更して使うこともできる。

４．動的複合実行方式

以下では、ソースコードでの実行を、それが必要な時に必要な部品だけに適用するためにＤＨＥ方式をどのように用いるかについて述べる。ここで、部品とは単体の手続きまたはデータ（ＰＡ）、さらに両者の集合としてのプログラム・モジュール（ＰＭ）を指す。ある部品がソースコード、オブジェクト・コードのいずれとして実行または参照されるかを、その部品の実行モードと呼ぶ。各部品には固有の実行モードが与えられ、そのモードで実行される。各部品の実行モードはプログラマや実行中のプログラム自身から随時設定、変更できる。ただし、これを常に明示的に行うのは繁雑なので、標準的な実行モードの設定と切り換えの機能を用意し、明示的な指定がなされないときにはこれに従うこととする。

４．１　自動的な実行モードの切り換え

（１）ソースモードへの切り換え

テストやデバッグを高速化するためには、誤りの含まれる可能性の高い部品だけをソースモードとして実行を開始し、他の部品は実行時にその部品から影響を受けた時点で、ソースモードに変える。そのために、誤りを含む、またはその可能性の高い部品に印をつけておき、実行時にこの部品から参照された部品およびこの部品を参照した部品の実行モードをＤＬＬが動的リンキング時にソースモードに変える。また、この部品から書き込みが行われたデータに印をつけ、これを読み出した部品をＤＨＥＫがソースモードに変える。このようにして、実行モードが伝播される。このように実行モードの切り換えが動的に行われることによって、誤りの影響が実際に及ぶまでは、プログラムはオブジェクトモードで高速に実行されるので、より高速に、問題のある箇所にたどり着くことができる。

（２）オブジェクトモードへの切り換え

誤りを含む部品を全く特定できないときは、全ての部品をソースモードとして実行を開始し、信頼性の高い部品から順にオブジェクト・モードに切り換えてゆく方法をとる。ソースモードでの実行時には、テスト・カバレッジ情報や部品間の参照回数情報の収集を行い、この情報を利用して、部品のカバレッジや参照回数がユーザの指定した基準に達したときに、その部品をオブジェクトモードに切り換える。この方式は、（１）の方式で連鎖的にソースモードに切り換えられた部品を、再びオブジェクトモードに戻す場合にも有効である。

４．２　テスト時の動的複合実行

テストの段階での目標はできる限り多くの誤りを検出することであり［１１］、会話性よりも、一定時間により多くのテストケースを高速に消化できることが重要であるので、オブジェクトモードでの実行を基本とする。ただし、テストケースは通常プログラムの論理に注目して作られるので、逆に言語の意味レベルで規定される単純な誤りがたまたまテストの網の目をくぐり抜けてしまうことがあり、これはしばしば難解なバグとなって後に発現する。これを防ぐため、テスト時にも以下のように必要に応じてソースモードでの実行を組み合わせ、このような誤りのチェックを自動的に行う。

ボトムアップまたはトップダウンに開発を行っている場合、発見された誤りは最後に加えられた部品にその原因がある可能性が高い。同様に逐次的な変更を加えていった時に発見された誤りは、最後に変更された部品が原因となっている可能性が高い。すなわち、ある時点でプログラムに誤りが含まれないなら、その後の部品の追加または変更によって生じる誤りの原因となっている部品は、追加または変更された部品あるいはそれによって影響を受ける部品である。このような影響の及ぶ範囲（部品の集合）をテストのスコープと呼ぶ。テストにおいては、テストのスコープに含まれる部品だけをソースコードでの実行の対象とするために、エディタによって変更された部品に印をつけ、４．１で述べたような方法で実行モードの伝播を行う。

４．３　デバッグ時の動的複合実行

エラーを追い詰めて行くデバッグの過程では、明示的かつ詳細なモードの切り換えが必要である。デバッグ時にソースコードでの実行が及ぶ範囲を、デバッグのスコープと呼ぶ。最初のデバッグのスコープは、テストのスコープと同一であり、このスコープ内でエラーが発見される可能性は高い。誤りの所在が追い詰められて行くに従ってデバッグのスコープをせばめてゆくことにより、デバッグを効率化することができる。デバッグのスコープの縮小を支援するために、次のような実行モードの切り換えの機能を提供する。

（１）　ランダムな個別指定

誤りを含む、または含む可能性の高い部品だけを選択的にソースモードで実行する。

（２）　実行の履歴による指定

ある部品が、エラーを引き起こすまたはその可能性の高い特定の呼び出し（参照）の経路を経て参照されたときに限り、またはそれ以降、ソースモードで実行する。参照経路の表現には、一般化された順路式［７］を用いる。

（３）　事象の発生による指定

エラーを引き起こす可能性の高い事象の発生または条件が成立したときに限り、またはそれ以降、ある部品をソースモードで実行する。

これらの指定方法を用いた場合にも、４．１に述べたような実行モードの伝播が適用される。

一方、過去の変更で入り込んだ誤りが後に発現した場合、それを突き止めるのはより困難な仕事である。ただし、この場合にもこの誤りを誘発した要因は、現在のテストのスコープにある。したがって、デバッグのスコープは現在のテストのスコープを始点として徐々に拡大され、手がかりが摑めた時点から再び縮小される。誤りの所在が確定された後は、その誤りが混入した時点のテストのスコープまで戻り、正しく変更を行った後、過去のテストのスコープとそこでのテストをたどりながら、現在のテストのスコープまで回復する。

５．動的リンキング方式

動的リンキング方式は、頻繁に変更され、かつ繰り返し実行されることの少ないプログラムに対するリンキングのコストを削減する。開発時のプログラム、特にデバッグ中のプログラムはその典型的なものである。ＩＰＳでは、動的リンキングを行うとともに、これを利用して動的なブロック構造の形成を行う。

５．１　部品の動的なリンキング

動的なリンキングでは、実際には参照されない部品はロード、リンクされる必要がないため、大きなプログラムの一部分だけの変更、テスト、デバッグのサイクルが効率化される。また、実行時に実際に参照されない部品は、例えプログラムの中に呼び出しが書かれていても実在する必要がなく、スタブを明示的につくる必要がない。ＩＰＳでは、部品間の参照は全てＤＨＥカーネルを介して行われるので、この特徴を積極的に利用してシンボリックな動的リンキングを行う。すなわち、各部品からの外部参照は被参照ＰＡの名前で表現し、実行時にこれが実際に参照されたとき、名前を実体に結合する。

５．２　ブロック構造の動的な形成

一般にブロック構造を持つプログラムでは、実行前にブロック構造が完成している必要があり、例えば、下位のブロックだけを独立にテスト、デバッグすることが困難である。ＩＰＳでは、動的リンキングの機能をさらに有効に利用して、不完全なブロック構造からの実行開始を可能にし、必要に応じてブロック構造の動的な形成を行う。これはまた、Ｃのようにブロック構造を持たない言語に対しても、ブロック構造による名前のスコープの制御の機能を与える。

６．会話型プログラミングの例

以下では、簡単な例を用いてＩＰＳ上でのプログラミングの過程を説明する。ＩＰＳ上では、プログラムの作成と変更、実行とデバッグは、プログラミングの過程のそれぞれ独立した段階ではなく、相互に絡み合った一連の作業である。ソースプログラムは、プログラム実行の途中で必要となって作成されたり、デバッグの途中で”パッチ”される形で変更されてもよい。また、ソースプログラムは入力と同時に、ソースコード・インタプリタで実行可能であり、即時にテストやデバッグに移行できる。

６．１　ソースプログラムの作成と変更

ソースプログラムの入力は、構造エディタＥＤＴを用いて行われ、構文誤りが入力時に防止されるとともに、ソースコード・インタプリタで実行可能な木構造表現が入力と同時に生成される。

（１）テキストの入力

構造エディタの入力方式は、構文規則のテンプレートを用いた合成（ｓｙｎｔｈｅｓｉｓ）型と、一次元的な入力に対する構文解析（ｐａｒｓｉｎｇ）型に分類できるが、入力のし易さや初心者、熟練者の何れに向くかなどの点で両者ともに一長一短である［１２］。そこでＥＤＴではこの両者を融合した形式を用いる。ＥＤＴにおけるテンプレートは入力者の明示的な操作によってではなく、一次元的なテキスト入力に対する構文解析の結果、ＥＤＴから生成され、入力者の確認によってテキスト中に取り込まれる。すなわち、ソースプログラムは入力と同時に構文解析され、現時点以降の構文が一意に推定できる状態になったとき、構文の残りの部分のテンプレートが自動的に付加される。その後の入力は、このテンプレートをなぞりながら行われるので、誤った構文は作られない。この方式では、通常のテキスト・エディタ上でプログラムを作成する場合とほぼ同様なキー操作で、テキストを入力することができるとともに、既存のソースコード・ファイルをエディタを介して取り込むことができる。

以下、図２のようなプログラムの一部分を入力している場合を例に、プログラムの入力方法を説明する。入力されているソース・ステートメントの構文が一意に推定されるまでは、入力は”未確定”文字列として、高輝度表示される(i)。未確定状態での入力誤りは、削除キーによって自由に取り消すことができる。キーボードから、”ｆｏｒ（”までの文字が入力された時点で、これがｆｏｒ文であることは一意に定まるので、エディタは(ii)のように制御構造の残りの部分を付加表示して、それ以降の入力をｆｏｒ文として解釈することをユーザに知らせる。もしこれが、ユーザの意図と異なるとき、例えば”ｆｏｒｋ（）”と入力しようとしていたなら、すぐに誤りに気づく。この場合、削除キーを打鍵することによって（ｉ）の状態に戻すことができる。一方、もともとの意図に合致していたなら、ユーザはこれを承認し、それ以前の入力を”確定”する。付加された部分はそれ以降、テンプレートとしての役割を果たす。(iii)の場合、ユーザの承認は、それ以降のソースコードの入力をそのまま続けることによって示されるている。その後、テンプレート中にソースコードを埋め込み(iv)、残りのテンプレートをなぞると、そのソースコードが確定される(v)。以下、同様にして入力が続けられる。

	ユーザ入力	表示
i	ｆｏｒ	ｆｏｒ＿
ii	（	ｆｏｒ（＿；）；
iii	ｉ	ｆｏｒ（ｉ＿；）；
iv	＝０	ｆｏｒ（ｉ＝０＿；）；
v	；	ｆｏｒ（ｉ＝０；＿）；

図２．テキスト入力例

（２）　誤りの検出

テンプレートとして入力されたテキストには、構文誤りは存在しない。一方、テンプレートが用いられない式の誤りには、それが誤りと認識される最初の文字の入力直後に、エラー・メッセージがディスプレイに表示される。このとき、この誤りは訂正されてからでないと入力の継続は許されないので、結局、構文誤りを含むプログラムは作成されない。未定義な名前の使用やデータ型の不一致などの意味的な誤りに対しては、警告メッセージが出される。これを受けて、ユーザは現在の入力を未確定のまま一時中断し、データの定義を調べたり、変更したりした後、この入力を再開することができる。意味的な誤りを含む文字列を確定することはできない。

（３）　スタブ

未確定な部分から図３(ii)のようにカーソル移動によって脱出し、他の部分の入力に移行することができる(iii)。この場合未確定のまま残された部分は”スタブ”となり、高輝度表示される。スタブに対しては、後にカーソルをここに移動して入力を継続することができる。実行時にこの部分に達したときには、エディタが自動的に呼び出され、カーソルがこのスタブを指す。このときも同様に入力を継続し、この部分を確定した後、実行が再開される。プログラミングの初心者のために、現在のカーソル位置にあるスタブに入るべき構文を表示する窓を画面上にとることもできる。

	ユーザ入力	表示
i		for（i＝0；□）；
ii	カーソル移動キー	for（i＝0；□；）；
iii	i＋＋	for（i＝0；□；i＋＋）；

図３．スタブ生成の例

（４）　隠蔽

ソースコードを表示する際、現在見る必要のない部分を隠蔽することができる。これは、データ宣言部分と現在追加や変更、デバッグの対象となっている部分だけを表示するというように利用する。また、デバッグ用ステートメントの隠蔽、コメントの隠蔽、トラップが設定されている部分だけの表示などの形でも利用できる。他の部分の変更、例えばデータ宣言の変更や、ラベルの削除などによって影響を受ける部分の隠蔽は自動的に解かれる。

（５）　修正

一度確定してしまったテキストに修正を加えるには、修正したい部分にカーソルを移動し、削除キーを打鍵すると、その文字の修正に影響を受ける部分が未確定に戻される。未確定化された部分を、変更し確定する手続きは、その入力時と同様である。データ宣言部分の変更は、それを参照しているステートメントを全て未確定に戻す。

６．２　プログラムの実行と制御

プログラムを実行するためには、ユーザコマンド・インタプリタＵＣＩに、手続き呼び出しのためのステートメントを与えればよい。このステートメントは、現在停止しているユーザプログラムの手続きのスコープ内で、即座に実行される。

（１）　トラッピング

プログラムの実行は任意の時点でトラップすることができる。制御の流れだけでなく、データに対するアクセスに対するトラップを設定することが可能である。このトラップには、その発生時にどのようなアクション（トラップ処理手続き）を行うかが指定でき、一般の”ブレークポイント”のように制御を端末に戻す操作は、そのアクションの一つとして扱われる。トラップ処理は、通常のプログラムと同様な任意のソース・ステートメント列で書くことができるので、プログラマはそのための特別な言語を学習する必要はない。

トラップを設定するには、図４のようにソースコードが表示されている画面上で、トラップを設定したい部分にカーソルを移動し(i)、トラップ挿入コマンドを入力すると、(ii)のようにトラップ設定用の窓が表われる。この窓にトラップ処理手続きを、６．１で述べたような方法で書き込む(iii)。空のステートメントまたは明示的なユーザコマンド・インタプリタへの呼び出し”ＵＣＩ（）”が書かれると、そのトラップ発生時にＵＣＩが起動される。ＵＣＩからエディタを起動すると、画面上のカーソルはこのトラップを指す。エディタでソースコードの修正を行ったり、新たなトラップを設定したりした後、ＵＣＩを終了すると、プログラムの実行が再開される。また、図５のように宣言部分にトラップを設定すると、そのデータに対するアクセスが行われたときに、トラップ処理手続きが起動される。トラップ処理手続きは、後に削除するか、文法的に許されるものに対しては括弧を外して、通常のコードに変えることができる。

	表示
i	x＝y／z；
ii	<≥x＝y／z；
iii	<if (z==0){ printf (”0 divide”); UCI ();} ≥x＝y／z；

図４．トラップ設定の例

	表示
i	int cnt；
ii	int <≥cnt；
iii	int <printf (”counter=%d¥n”, cnt); ≥cnt；

図５．データに対するトラップの設定の例

i	a = b + c * d； <a = 1 + 2 * 3>
ii	a = b + c * d； <a = 1 + 6>
iii	a = b + c * d； <a = 7>

図６．　式の１ステップ実行の例

（２）　１ステップ実行

プログラムのうちソースモードで実行されている部分は、会話的に１ステップづつ実行し、その動作を追うとともに、各ステップでプログラムの実行状態の観察や変更、さらにソースコードの変更を行うことができる。１ステップ実行では、指定された歩幅で１ステップが実行され、実行がどこまで進んだかが画面のソースコード上にカーソルで示される。１ステップの歩幅はソースコードの１トークンごとを最小歩幅として、オペレーション単位、ステートメント単位、手続き単位が選択できる。図６に、代入文をオペレーション単位で１ステップづつ実行するときの様子を示す。式の評価に対しては、現在その式の評価がどこまで進んでいるかを見やすくするために、式の評価の過程を表わす窓に図のような表示を行う。さらに、１ステップ実行を自動的に連続して行うことによって、プログラムの実行の流れを画面上で視覚的にトレースすることができる。この場合、１ステップの歩幅だけでなく、制御構造のどのネストレベルまでをトレース対象とするかを指定することができる。

（３）　実行状態の変更

トラップや１ステップ実行によって停止しているプログラムに対して、データの値を調べたり、データを書き換えたり、制御の流れを移したりするには、通常のプログラム中でそれを行うために書かれるステートメントを、キーボードから直接入力すればよ

い。ユーザは、現在停止している手続きの中のスコープにおり、キーボードから入力されたステートメント列は、トラップ発生の直後に一時的に挿入されたステートメント列として実行される。

（4） 実行モード

現在のデバッグの対象に含まれない部品は、オブジェクトモードで高速に実行される。例えば、トレースのネストレベルが指定されているときには、そのレベル以下で呼び出される手続きは全て、また、一つの手続き内のみに注目しているときは、それ以外の手続きは全て、オブジェクト・モードで実行される。また、ユーザが明示的に指定した手続きやデータに影響を受ける手続き、影響を与える手続きだけをソースコードで実行するように指定するなど、4章で述べたような様々な実行モードが選択できる。

7．おわりに

プログラミング環境において高水準な会話性と実行の高速性を両立するための基礎としてＤＨＥ方式を提案し、これを基礎とする統合的なプログラミング・システム、ＩＰＳを開発中である。ＩＰＳ上では、デバッグの対象となる部分はソースコード・イメージでインタプリットされるため、プログラムの実行状態の制御、把握、変更が容易であり、またデバッグ対象でない部分はオブジェクトコードとして高速に実行されるのでプログラム全体としての実行は高速に保たれる。これによって、従来の計算機上でも高速なソースレベル実行のイメージをユーザに与えることができるので、本方式は、実際の”実用的”なプログラムの開発に十分役立つと考えられる。現在、ＶＡＸ１１／ＵＮＩＸ上で開発を行っているが、最終的には、ＵＮＩＸシステムに依存しない実時間プログラムやシステム・プログラムの開発にも利用できる環境の実現を目標としている。

謝辞 最後に、本研究を行うにあたって御指導いただいた中田育男教授、益田隆司教授、佐々政孝講師、清木康講師に感謝致します。

参考文献

1）Chu,Y. and Itano,K.: Interactive Direct-execution Programming and Testing, Proceeding of COMPSAC'82 (1982).

2）Teitelman,W.: The Interlisp Programming Environment, COMPUTER, Vol.14, No.4, pp.25-33 (1981).

3）Medina-Mora,R. and Feiler,P.H.: An Incremental Programming Environment, IEEE Trans. on Soft. Eng., Vol.SE-7, No.3, pp.472-482 (1981).

4）Fritzson,P.: Preliminary Experience from the DICE system: A Distributed Incremental Compiling Environment, ACM Soft. Eng. Notes, Vol.9, No.3, pp.472-122 (1984).

5）Teitelbaum,T. et al.: The Why and Wherefore of the Cornell Program Synthsizer, SIGPLAN Notices,Vol.16, No.6, pp.8-16(1981).

6）佐藤豊，板野肯三：動的複合実行方式，ﾃｸﾆｶﾙﾉｰﾄ HLLA-33,筑波大学電子情報工学系(1984).

7）Bruegge,B. and Hibbard,P.: Generalized Path Expressions: A High Level Debugging Mechanism, SIGPLAN Notices, Vol.18, No.8, pp.34-44 (1983).

8）UNIX Programmer's manual, Department of Electrical Engineering and Computer Science, Univ. of California, Berkeley (1983).

9）Myers,G.J.: Advances in Computer Architecture, John Wiley & Sons,Inc. (1978).

10）Itano,K.: Structured Source Program Representation for Pascal Subset, Technical Note HLLA-24, Institute of Information Sciences and Electronics, Univ. of Tsukuba (1984).

11）Myers,G.J.: Software Reliability Principles and Practices,John Wiley & Sons, Inc.(1976).

12）原田賢一：構造エディタ，情報処理，Vol.25, No.8,pp.767-776(1984).