高信頼性ソフトウェアの開発を支援する計算機システム

A COMPUTER SYSTEM FOR RELIABLE SOFTWARE DEVELOPMENT

坪谷　英昭 Hideaki Tsubotani　平井　寿美 Toshimi Hirai　門田　憲明 Noriaki Monden
西　直樹 Naoki Nishi　田中　稔 Minoru Tanaka　市川　忠男 Tadao Ichikawa

広島大学　工学部
Faculty of Engineering, Hiroshima University

1. はじめに

プログラミング言語においては，ソフトウェアの信頼性を高める目的で，

- 抽象データ型を含み，ユーザが任意にデータ型を定義できる機能
- 例外処理機能

などが仕様にとり入れられ（Clu[1],[2], Ada[3] など），実際にその処理系が作成されているものもある。

言語がこのように高機能化しているにもかかわらず，これらの言語で書かれたプログラムを実行する計算機アーキテクチャ（命令セットのレベル）は，依然として従来のフォン・ノイマン型に基づいたものである。

このため，次のような問題が起こっている。

(i) コンパイラなどのシステム・ソフトウェアに大きな負荷がかかる。

(ii) このような計算機システム上で開発されるソフトウェアの信頼性，および生産性の低下を招く。

本稿では，まずプログラムの実行時におけるエラーをアーキテクチャのレベルで検出する計算機アーキテクチャSPRING（Software Priciples Integrated Architecture）[4]の概略を説明し，次に，このアーキテクチャをソフトウェアの開発環境にとり入れたソフトウェア開発支援システムの構築について検討する。

研究目的は，次の2点である。

1) 信頼性の高いソフトウェアを効率よく開発できるプログラミング環境の実現。

2) システム・ソフトウェアの負荷の低減。

ここで言う計算機システムとは，次のものを含む。

- SPRINGアーキテクチャ
- オペレーティング・システム
- プログラム開発支援ソフトウェア（コンパイラ，リンカ，デバッガetc）

2. ソフトウェアの信頼性および生産性

2.1 ソフトウェアの信頼性

一般にソフトウェアの信頼性が高いと言うことは、そのソフトウェアが要求仕様通りに動作することを意味する。ソフトウェアの信頼性向上のためにプログラムを高級言語で書くことは，これが単に望ましいというだけではなく，変更，保守の際のコスト低減という点からも必要なことである。また，プログラミング言語においても，抽象データ型や例外処理など，ソフトウェアの信頼性向上に役立つ機能を備えた高級言語が設計され，その処理系まで作成されているものもある。

これらの言語は，次のような特徴をもつ。

```
declare
  subtype NATURAL is INTEGER
              range 1..INTEGER'LAST;
  I: INTEGER;
  N: NATURAL;
begin
  N := I - N; --subtract operation is legal,
              --but assignment operation raises
              --constraint error for I-N<=0.
end;
```

図1. 副型の制約

```
declare
  subtype NATURAL is INTEGER
              range 1..INTEGER'LAST;
  type STRING is array(NATURAL range<>)
                   of CHARACTER;
  X: STRING(1..10);
  Y: STRING(1..N);
begin
  X := Y;
end;
```

図2. 添字の制約

(1) コンパイル時に静的に定められない要因を数多く持つ。

(2) 言語処理系における負担が大きい。

(3) 言語の設計理念に信頼度の高いソフトウェアの作成を支援することがもりこまれている。

(4) システム記述言語としても用いることができる。

(1)により，これらの言語で書かれたプログラムの信頼性向上のためには，プログラム実行時に種々のチェックを行なわなければならない。実行時チェックの例をAdaで示す。図1は，副型に課せられた制約のチェックは実行時になされなければならないことを示している。図2は，制約それ自身が実行時に定まる場合である。この例の場合，Yの添字の範囲制約は実行時に決定される。すなわち，添字の上限値は，Yの宣言が評価される時点でのNの値に依存する。YのXへの代入は，Yが添字範囲1..10を持つ場合にのみ実行される。

ソフトウェアの信頼性向上のためには，Adaのような高級言語で書かれたプログラムの実行時に，次のような事項が行なわれることが望まれる。

① 実行時エラーの発見

② 実行時エラーによる波及効果の防止

③ 実行時エラーからの回復

④ 実行時エラーに関する情報の記録と提供（デバッグのため）

実行時エラーを発見するためのチェックは，通常，コンパイラによって生成されるチェックコードを付加することによって行なわれている。しかしこの方法は，次のような欠点を持つ。

i) プログラム実行速度の低下

ii) コンパイラの負担の増大

また，こうして開発されたソフトウェアが実際に運用される時点では，実行速度の向上のためにチェックコードが取り除かれることがしばしばある。これは，ソフトウェアの信頼性低下の大きな要因であると考えられる。

第3章において我々は，ソフトウェアの信頼性，特に実行時の信頼性の向上のために，種々の実行時チェックをアーキテクチャのレベルで行なう計算機アーキテクチャを提案する。

2.2 ソフトウェアの生産性

通常のソフトウェア開発プロジェクトにおいては，システムの分析と設計，およびプログラミングが終了したあとのテスト・デバッグ段階が，全体のコストの40～50%を占めていると言われる。[5] 従って，この段階の生産性を高めることは，ソフトウェアの生産性の向上に大きく貢献する。また，使用するツールの有用性が生産性に大きく影響する。

テスト時に用いられるツールは，プログラムを実際に動かすかどうかによって静的テスト用のものと動的テスト用のものとに分類される。

静的テストには、

- ・コード監査
- ・静的解析
- ・記号実行・評価

などがあり，動的テスト用ツールには，

- ・テスト・データファイル生成
- ・自己計測型インストルメンタ
- ・動的表明チェッカ

などがある。

第４章において，アーキテクチャのサポートによる高度なデバッギング・ツールの実現について検討する。

3．SPRINGアーキテクチャ

本章では，アーキテクチャのレベルでプログラム実行時に種々のチェックを行ない，実行時信頼性の向上を支援するアーキテクチャSPRINGについて述べる。このアーキテクチャが対象とする言語は，Pascal，Adaなどの Algol系の言語である。

3．1　設計原理

SPRINGの設計において採用した原理は，次の４点である。

(1)　インストラクション・レベルでは，アドレスの概念を用いない。

(2)　プログラム中で宣言されるデータ型を認識する。

(3)　データから独立した命令体系を持つ。

(4)　モジュールの独立性を高める。

3．2　写像方式

次に，SPRINGにおけるオブジェクトの写像方式について説明する[6]。写像方式の概要を図３に示す。オブジェクトの写像を行なうために，数種類のディスクリプタを用いる。各ディスクリプタの書式を図４から図８に示す。各

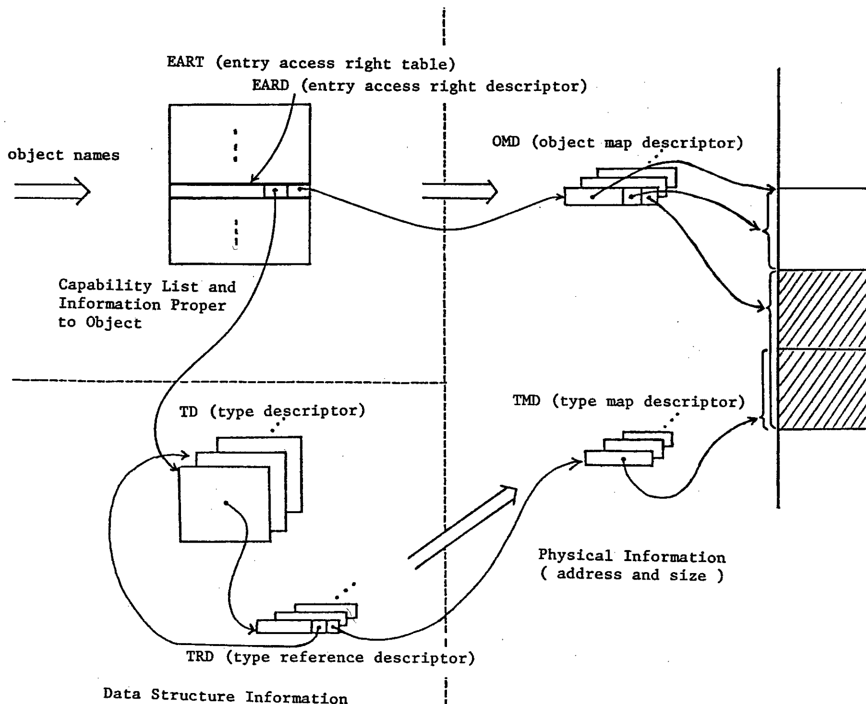


図３．　写像方式の概要

ディスクリプタに保持される情報は次のとうりである。

(a) EARD (Entry Access Right Descriptor)

宣言されたオブジェクトのそれぞれに対してEARDが個別に用意される。EARDに保持される情報は，それぞれのオブジェクトに依存した情報である。

EARDの集合であるEARTは，プログラム中から可視なオブジェクトの集合を定義している。EARTは，実行時にはコンテキストによって保持され，実行環境を構成する。オブジェクトのアクセスに際しては，最初にこのEARTが参照される。

(b) TRD (Type Reference Descriptor)

TRDは，TDとともにデータ構造の情報を保持するために用いられる。TRDの書式は，オブジェクト識別子がない以外はEARDと同じである。

(c) TD (Type Descriptor)

言語レベルで定義される基本となるデータ型は，スカラ型(整数型，実数型，論理型，文字型，列挙型)，構造型(配列型，レコード型)，およびアクセス型である。このようなデータ型を表現するためにTDが用いられる。TDの書式は各データ型によって異なっている。

(d) OMD (Object Map Descriptor)

OMDは，オブジェクトを計算機の記憶域に写像するための物理的な情報を保持している。各OMDは，EARDからポイントされている。

(e) TMD (Type Map Descriptor)

TMDは，プログラム中で定義された型を写像する際に必要となる記憶域の大きさを保持している。

3.3 実行時チェック

このアーキテクチャでは，オブジェクトに対してのアクセスと演算はスタックを用いて行なわれる。スタックに積まれるのはオブジェクトに関する情報である。

命令実行時に，これらの情報を用いて種々のチェックが内部的に行なわれる。実行時チェックの内容とそれが行なわれる時点の違いから，チェックは次の4つに分類される。

(i) 可視性のチェック

アクセス要求に対して最初にチェックされることは，現在の実行環境のもとで，要求されたオブジェクトに対するアクセスが許されるかどうかということである。このチェックはEARTを用いることによって実現される。

(ii) アクセスの種類(read/write)に対するチェック

我々の写像方式では，EART中に目的とするEARDが存在すれば，そのオブジェクトに対する読み出しは許可される。これに対し，書き込みの可否はEARD中の定数フラグにより決定される。

(iii) 演算オペレーションに対するチェック

これは，演算命令に対するオペランドの型の整合性のチェックである。

(iv) 代入オペレーションに対するチェック

代入に対しては，(iii)で述べたチェックの他に制約に対するチェックが必要となる。このチェックは構造型のオブジェクトに対しては複雑なものとなる。

4. ソフトウェア開発環境

本章では，高信頼性ソフトウェアの開発を支援するためのシステム・ソフトウェア，特にコンパイラとデバッギング・ツールについて述べる。

4.1 コンパイラ

SPRING用コンパイラは，図9のようなオブジェクトを生成する。これらのオブジェクトは，次の3つのクラスに分類される。

object identifier	type identifier	derived type flag	subtype flag	prototype identifier	constant flag	check flag	pointer to TD	pointer to OMD

図4. EARD (Entry Access Right Descriptor) の書式

type identifier	derived type flag	subtype flag	prototype identifier	constant flag	pointer to TD	pointer to TMD

図5. TRD (Type Reference Descriptor) の書式

スカラ型

整数型

```
INT
dynamic flag
lower bound   (value or object name)
dynamic flag
upper bound   (value or object name)
```

実数型

```
REAL  (fixed or float)
accuracy constraint
        ⋮
dynamic flag
lower bound   (value or object name)
dynamic flag
upper bound   (value or object name)
```

論理型

```
BOOL
```

文字型

```
CHARACTER
dynamic flag
lower bound   (value or object name)
dynamic flag
upper bound   (value or object name)
```

列挙型(識別子)

```
I_ENUM  (identifier enumeration)
dynamic flag
lower bound   (value or object name)
dynamic flag
upper bound   (value or object name)
```

列挙型(文字表記)

```
C_ENUM  (character enumeration)
dynamic flag
lower bound   (value or object name)
dynamic flag
upper bound   (value or object name)
n  (entry number)
value (1)
value (2)
   ⋮
value (n)
```

構造型

配列型

```
ARRAY
element type
n  (dimensional number)
index type (1)
index type (2)
   ⋮
index type (n)
```

レコード型

```
RECORD
n  (component number)
component identifier (1)
component identifier (2)
   ⋮
component identifier (n)
```

アクセス型

```
ACCESS
access type
```

図6. TD (Type Descriptor) の書式

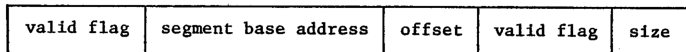


図7. OMD (Object Map Descriptor) の書式

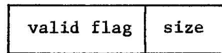


図8. TMD (Type Map Descriptor) の書式

(i) クラスI

このクラスのオブジェクトは，SPRINGによる実行のために必要とされるものである。これらは，従来のアーキテクチャが利用するオブジェクト・プログラムに対応する。

(ii) クラスII

このクラスのオブジェクトは，SPRINGのオペレーティング・システムやリンカーによって利用される。

(iii) クラスIII

このクラスのオブジェクトは，実行時エラーの処理のための情報の提供およびデバッギング・ツールによって利用される。

4.2 デバッギング・ツール

4.2.1 高水準デバッギング

通常，ツールを用いてのデバッギングは，プログラム中のある場所にブレイクポイント（中

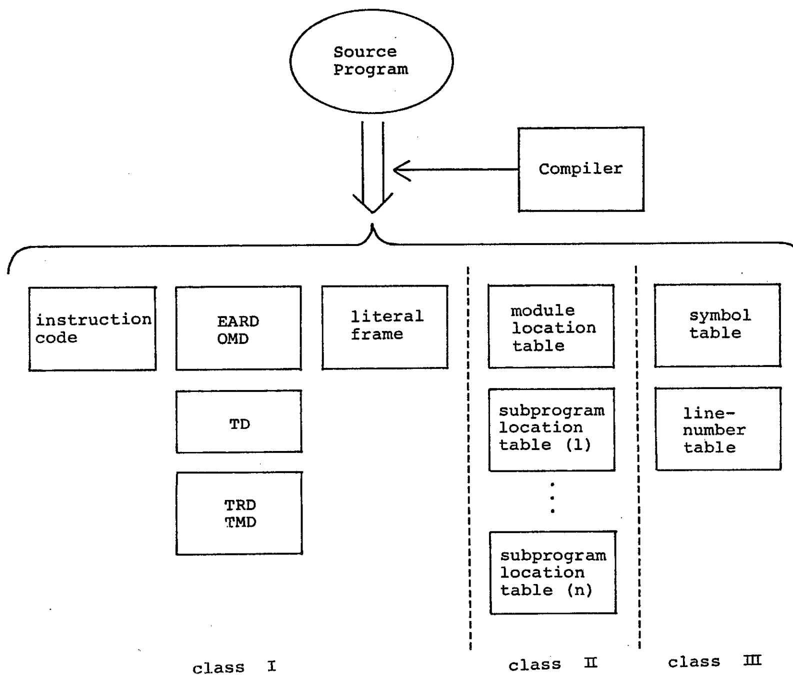


図9. コンパイレーション

断点）を設定し，実行がその場所に達した時プログラムを中断してプログラムの状態に関しての情報を得る，というようにして行なわれる。

このようなデバッギングでは，次のような問題がある。

(1) プログラマは，適切にブレイクポイントを設定するために，そのプログラムの実行順序の詳細を理解しなければならない。

(2) 人と計算機システムとの間で交わされる情報は広範囲にわたり，出力される情報はほとんどformattingされていない。（メモリダンプなど）

このような低水準のデバッギングに対して，高級言語で書かれたプログラムのデバッグ作業の効率を高めるためには，次のような条件が満たされなければならないと考える。(7)

(i) プログラムを中断させるためのブレイクポイントは，特定のコントロールポイントに達した時よりも，むしろプログラムの状態に関するある条件が発生した時に連想的にセットされる。

(ii) デバッギング・コマンドは，プログラミング言語のソースレベルで与えることができる。

4.2.2 従来のデバッギング・ツール

従来行なわれているデバッギングを図10に示す。

(a)は，デバッグのためのツールを用いない場合である。プログラムの適当な箇所に入出力文を挿入してプログラムを走らせ，変数の値など，プログラムの状態に関する情報を得る。このようなデバッグにおいては，バグが発見できるまでプログラムをコンパイルして走らせることを繰り返さなくてはならず，デバッグの効率は非常に悪い。

(b)は，動的表明チェッカと呼ばれるツールを用いる場合である。プログラマは，プログラム中の適当な箇所に，その場所において満たされていなければならない条件（表明，assertion）を論理式の形で記述した文を挿入する。これらの文はコメントの形で書かれるので，表明のチェックを行なう命令をソースプログラム中に挿

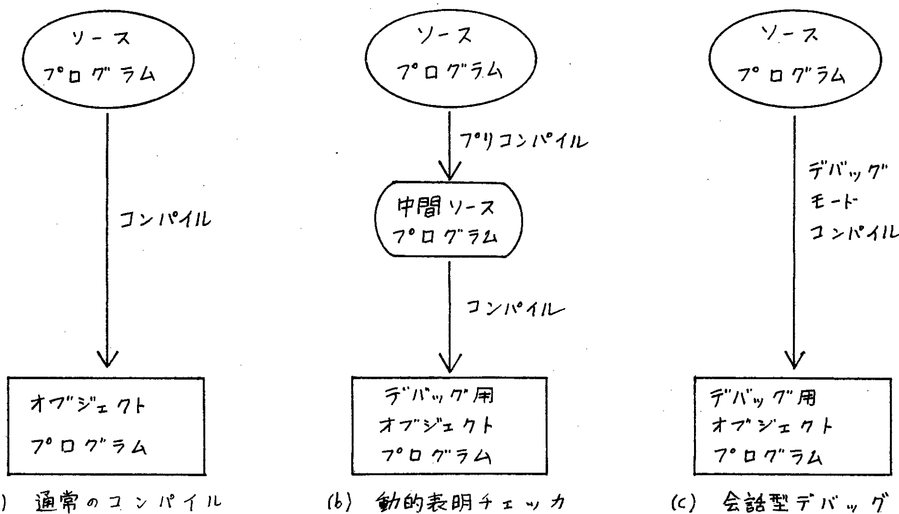


図10. 従来のデバッグ過程

入するためにプリコンパイルを行ない，その後で通常のコンパイルを行なう。このツールを使う場合，表明が成り立たない場合にのみプログラムが中断するのでデバッグの効率は向上するが，会話型デバッグのような柔軟性はない。

(C)は，会話型デバッギング・ツールを用いる場合である。プログラムをデバッグ・モードでコンパイルした後，プログラマは，会話的にプログラム中にブレイクポイントを設定あるいは削除してプログラムを走らせ，デバッグを行なう。このようなツールを用いることによってデバッグの効率は向上するが，プログラム実行時のオーバヘッドが大きくなる。

4.2.3 SPRINGデバッギング・ツール

4.2.1で述べたような高水準デバッギングのための要件を満たし，デバッグの効率向上を支援することを目的として現在開発中のツールについて述べる。

このデバッギング・ツールは，次のような特徴を持つ。

(i) ブレークポイントは，ソース・プログラム中のオブジェクトに制約を付加することによって設定する。これにより，試行中のプログラムは，あるコントロールポイントに達した時にではなく，プログラム中のオブジェクトが要求された制約を満たさなくなった時点でブレークする。

(ii) デバッグのためのプログラムの再コンパイルを必要としない。デバッギング・ツールはメモリ上のEARDだけを操作し，プログラムに制約のチェックを手続きの形で付加する。チェックのためのコードをオブジェクト・プログラムに挿入するという方法ではないので，プログラムの再コンパイルを行なう必要がない。

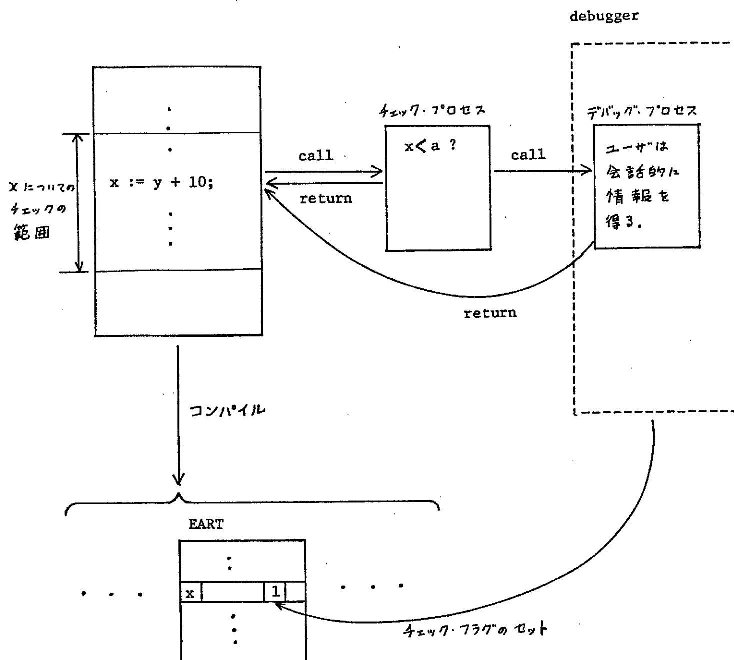


図11. デバッギングの概要

次に，このデバッギング・ツールを用いてデバッグがどのように行なわれるかを述べる。

例として，プログラム中の変数xに対して，15行から30行までの範囲で$x<a$という制約を課した場合について述べる。（図11参照）

① オブジェクトに対する制約の記述

制約を課す対象オブジェクト，制約，および制約を課すソース・プログラムの範囲

x

$x<a$

line 15..30

を記述する。

これにより，デバッギング・ツールはxのEARD中のデバッグ用チェックフラグをセットする。

② プログラムの実行

プログラム実行中，SPRINGは，オブジェクトへの代入命令（Store）に対して，そのEARDのチェックフラグがセットされていて，かつ制約の範囲に入っているかどうかを調べる。両方とも成り立てば，制約のチェックを行なうプロセスを呼び出す。このプロセスは制約をチェックし，制約が満たされていなければデバッグのためのプロセスを呼び出す。

③ 会話的デバッグ

ユーザは，デバッグ・プロセスで，システムから変数の値などの情報を会話的に得る。

5．実験システム

筆者らは，これまで述べたようなアーキテクチャとシステム・ソフトウェアをミニ・コンピュータHP-1000上に構築中である。

SPRINGは，論理的には，図12に示すようにメモリ共有結合でホストと結ばれている。実験システムの構成は図13のとうりである。

SPRING本体は3Kワード（1ワード/24 bit）のマイクロプログラムである。また，これを支援するオペレーティング・システムはホスト上で動作する64 K byteのプログラムで，入出力とメモリ管理を主に行なっている。

現在，SPRINGとOSの開発を終え，SPRING用Pascalコンパイラの試作およびデバッギング・ツールの設計を行なっている。

6．評価

SPRINGアーキテクチャ，およびこれを含む計算機システムの評価を，次の4点について行なう。

(1) プログラムの実行速度

実測データがまだ少ないので，定性的なことについて述べる。一般に，SPRINGのようなタグあるいはディスクリプタを用いるアーキテクチャでは，メモリ参照の回数が従来のアーキテクチャより増加する傾向がある。これによる実行速度の低下は，連想メモリ，キャッシュ・メモリ，パイプラインといったハードウェア技術を用いることによって改善

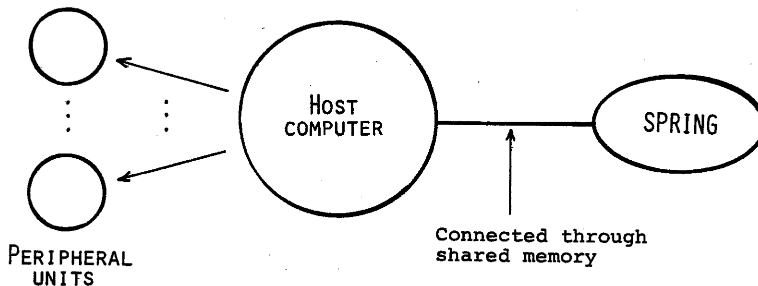


図12. SPRING実験システム

可能である。

SPRINGは，実行時に決定しなければならない要因の多い高級言語に対して有効であり，低レベルの高級言語，例えばFortranなどに対してはその性能をあまり発揮することはできない。

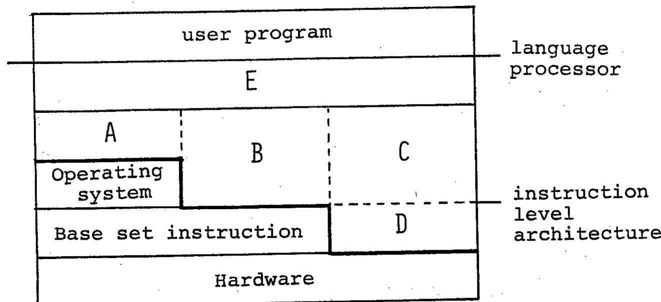
(2) システム・プログラムの負荷低減

ここではコンパイラについて考える。図14に一般的なコンパイラの構成を示す[8]。コンパイラは，解析相において原始プログラムの解析を行ない，各種情報の表を作成する。つぎに，合成相において内部形式に変換された原始プログラムから各種情報の表を使って目的プログラムを作成する。この過程で複雑な処理を要するのは，意味分析（意味づけ）とコード生成の準備の部分である。

一方，SPRINGにおけるコンパイラでは，図15で示されるように合成相がなくなる。すなわち，解析相はほぼ一般のコンパイラに準ずるものであるが，内部形式のプログラムはほぼそのままインストラクション・コード・セグメントに，また各種情報の表がほぼそのまま各種ディスクリプタ・セグメントになる。このようにして，SPRINGでは，コンパイラの負荷低減をはかることができる。

(3) ソフトウェアの実行時信頼性の向上

2.1で述べたソフトウェアの信頼性向上のために望まれる事項について，①，②に対し



A : I/O control program
B : Memory management program
C : Emulator interface
D : Emulator
E : System program (compiler,linker etc.)

図13. 実験システムの構成

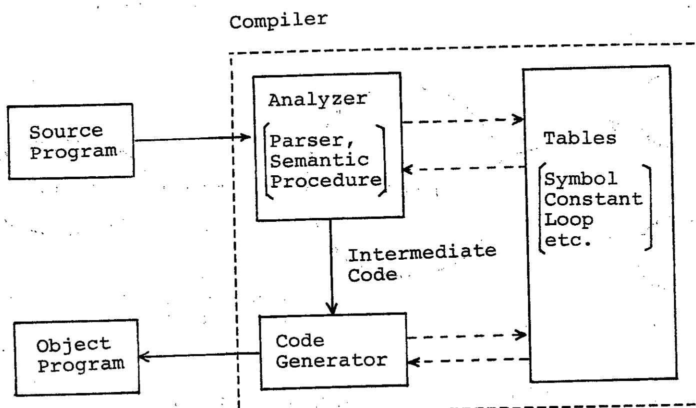


図14. 従来のコンパイラ

ではＳＰＲＩＮＧは十分な機能を提供していると考えている。

③に対しては，使用する言語とのかねあいもあり，ＳＰＲＩＮＧがどのような機能を持つべきかは未定である。

④に対しては，ＳＰＲＩＮＧはプログラマに十分な情報を会話的に提供する能力を保持している。ただし，今回の実験システムでは，実行時エラー検出後にソース・プログラム上でのエラーの位置とそのエラーの内容を表示し，プログラムを停止させることを基本機能と考えている。

(4) ソフトウェアの生産性の向上

プログラムのエラーのうち，静的なものはコンパイラによって検出され，動的なものはＳＰＲＩＮＧによって検出される。また，プログラムの論理的なエラーに対しては，システムの提供するデバッギング・ツールを用いることによって効率的にエラーを発見できる。これらにより，ソフトウェアの信頼性とともに，生産性も向上させることができると考える。

7. おわりに

本稿では，信頼性の高いソフトウェアの開発環境の実現，およびシステム・ソフトウェアの負荷の低減を目指してＳＰＲＩＮＧアーキテクチャを提案した。さらに，ソフトウェア開発を支援するためのコンパイラやデバッギング・ツールについても述べ，この計算機システムの評価を行なった。

ＳＰＲＩＮＧは，プログラムの実行時エラーを検出し，デバッグのための情報を提供することができる。このような計算機システムをソフトウェアの開発段階だけでなく実際の運用段階にも用いることは重要である。

今後は，さらにシステム全体の機能拡充と実行速度の向上を図り，システムの定量的評価を行ないたいと考えている。

謝辞

本研究において有益な討論と助言をいただいた本学情報システム研究室の方々に深く感謝いたします。

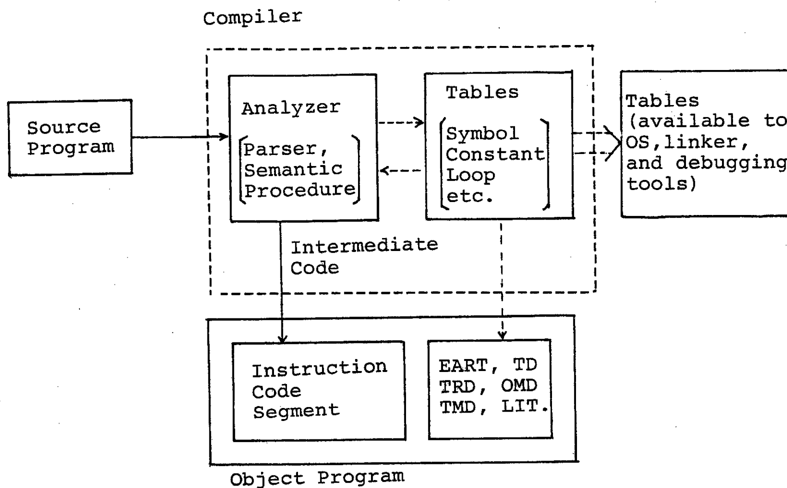


図15. ＳＰＲＩＮＧコンパイラ

（ 参 考 文 献 ）

(1) B.Liskov,A.Snyder,R.Atkinson,and C.Schaffert,"Abstraction Mechanisms in CLU," Communications of the ACM,Vol.20,No.8, pp.564-576,August 1977.

(2) B.H.Liskov and A.Snyder,"Exception Handling in CLU," IEEE Transactions on Software Engineering,Vol.SE-5,No.6, pp.546-558,November 1979.

(3) Reference Manual for the Ada Programming Language, United States Department of Defense,July 1980.

(4) N.Nishi,H.Tsubotani,and T.Ichikawa,"SPRING:A High Level Language Architecture in Ada Environment," Proceedings of the COMPSAC'83,pp.373-377,November 1983.

(5) Tutorial:Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers,IEEE,1980.

(6) 西直樹，市川忠男「データ型操作を支援するアーキテクチャについて」 信学技報 AL82-61，1982.

(7) D.Hamlet,"Debugging "Level":Step-Wise Debugging," SIGPLAN NOTICES,Vol.18,No.8.pp.4-8,March 1983.

(8) D.Gries,"Compiler Construction for Digital Computers," John Wiley & Sons,Inc.,1971.

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

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

(11) Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging,March 1983.

盛光印刷所