UDC 681.3.02

プログラム自動解析システム

Computer Aided Program Analysis System

斗納宏敏[1] 高橋 稔[2] 安木寿教[3]
Hirotoshi Tono Minoru Takahashi Hisanori Yasugi

内丸ひろみ[4]
Hiromi Uchimaru

要 旨

自動車制御へのマイクロコンピュータ応用の拡大とともに、制御プログラムの開発要員不足が深刻化している。制御プログラムでは高級言語の採用が困難であり、アセンブラ言語においても効率のよいプログラム開発技術が必要となる。プログラム設計の生産性を高めるにはプログラムのモジュール化や構造化技法の採用が有効であるが、プログラムのブラックボックス化による弊害が懸念される。

CAPASはアセンブラ言語で設計したプログラムを日常的な表現に置き換えてフローチャートと共に表すシステムであり、ブラックボックス化による問題点を取り除くことができる。また、経験の浅い設計者でも容易にプログラム内容が把握できるため、デバッグ作業の効率化・プログラムの信頼性向上といった効果がある。

With the increase of micro-computer's applications for automobiles, short of the programmers became serious. It is diffcult to use the high-level-language for control program, and an efficient programming method is requied to inprove the productivity. Module programming or stuctured method are effective to the productivity, but there are suspicion of bad effect that programs turn to black-box.

CAPAS is the system that analyzes the program designed by assembly language and generates flow charts with explanations as a results. That can resolve the problem in the cause of black-box, and makes it easer for beginners to understand how the program behaves, thus we have the effect, that is high efficiency of debugging and high reliability of a program.

(1)(4) モートロニクス本部技術部
(2)(3) 開発部

1. はじめに

自動車制御へのマイクロコンピュータ応用は、1970年代後半の排出ガス規制対策としてエンジン制御に用いられたのに始まる。現在ではトランスミッション、サスペンション、ブレーキといった直接性能に係わるものから、エアコン、オーディオ、ナビゲーションといったユーザの利便性を図るものまで、自動車のあらゆる箇所で利用されるようになっている。マイクロコンピュータの応用は、複雑な制御アルゴリズムを実現し、多様なユーザーニーズに対応することを可能とするため、その利用範囲は年々拡大している。一方、制御プログラムは大規模化・複雑化しており、開発要員の不足が深刻化している。

制御プログラムは処理能力やメモリ容量の制約から、アセンブラ言語を用いて開発するのが一般的である。マイクロコンピュータの高性能化とメモリの大容量化・低価格により、C言語などの高級言語によるプログラム開発も可能となってきたが、比較的小さなメモリ容量で、多数の入出力信号を高速に処理する必要があるエンジン制御では、全面的に高級言語を使用することは困難である。依然としてアセンブラ言語のしめる割合が大きく、アセンブラ言語を用いる場合においてもプログラム設計の生産性を向上させる技術の確立が急務となっている。

アセンブラによるプログラム設計の問題点はアセンブラ言語の難解さと、それに起因する多大なデバッグ工数によるところが大きい。本論文では、自動車制御におけるプログラム開発効率化の問題点を述べると共に、アセンブラ言語で記述されたプログラムの実行内容を解析・表示することで開発支援を行うシステムCAPASの紹介をおこなう。

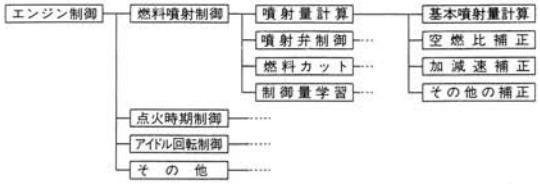
2. プログラム開発改善における問題点

2.1 設計方法の改善

ソフトウェアの生産技術として重要な概念に、「ウォータフォールモデル」と「構造化プログラミング」がある。前者はプログラム生産工程を、要求分析からシステム設計、プログラム設計、テスト、運用まで、上の工程から下の工程へと開発が進む様子を水の流れに例えてモデル化したものである。

後者はいわゆる「トップ・ダウン」式のプログラム開発を行う技法である。要求分析からプログラムの持つべき機能を大別し、大別した機能ブロックの中を更に小さな機能ブロックに分割するという手順を繰り返して設計を行う。分割された各プログラムブロックをモジュールと呼ぶ。モジュール化されたプログラムは独立した機能を有し、他のモジュールと独立した設計が可能であり、複数の設計者の共同作業によるプログラム開発を行うことができる。

これらの開発技法は、PascalやC言語といった構造化言語を使用するのが一般的であるが、基本的な概念はアセンブラ言語によるプログラム開発にも有効である。ただし、この技法を自動車制御の分野で利用するにはいくつかの解決すべき問題点がある。複合化が進んでいる自動車制御では、状況に機敏に応じるリアルタイム処理と、複数の機能を同時に実現する多重制御を行う必要があるが、構造化プログラミングでこれらの制御プログラムを設計することは困難である。構造化プログラミングを自動車制御に適用するには、各機能間で共通した処理をまとめて一つのモジュールとしたり、処理の優先度に応じて必要なタイミングに



モジュール・プログラミングでは、機能分割の過程がそのままプログラム設計の過程となることが基本であるが、エンジン制御ではモジュールの分割・配置に工夫が必要。

図－1　エンジン制御プログラムの構成図

Fig. 1　Engine control program

加減速補正はAD変換ルーチンの加減速判定、10ms定周期処理での補正量減衰計算などに処理の分割を行っている。

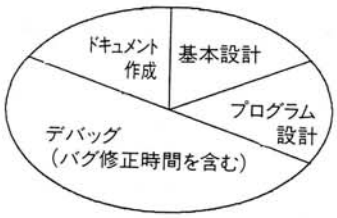
図－2　加減速補正計算のプログラムモジュール配置

Fig. 2　Program modules of compensation culculation

必要な処理が行えるよう処理内容を分割するなど、プログラム構成を工夫する必要がある。この場合、モジュール化されたプログラムの独立性が低くなり、他のモジュールとの関係を意識しながら設計を行う必要が生じてくる。各モジュールの理解度や制御仕様の理解度が大きく影響することにより、経験の浅い設計者ではモジュール設計が困難となる問題点が残る。図－1，2参照

2. 2　デバッグ方法の問題点

構造化プログラミング、あるいはモジュール・プログラミングは、プログラム開発を効率よく行うための技法であるが、開発したプログラムの検証に関してはふれていない。実際には開発したプログラムが要求仕様通りの処理をしているかを確認する作業が必要である。この作業をデバッグと呼んでいる。デバッグの結果、仕様に反した動作をしていれば、上流の工程に戻ってプログラムを修正する必要がある。この作業は要求仕様通りの動作をするまで何度となく繰り返すことになり、



図－3　プログラム開発の工数配分

Fig. 3　Tune distribution of program development

プログラムの開発工数で大きなウエイトを占めている。特に、自動車制御では小さなミスが重大な事故につながる可能性があるため、プログラムの動作確認は特に厳密に行う必要があり、全工数の50%以上をデバッグに費やしている。図－3参照

基本的なデバッグ作業は、仕様書に記述されている処理の実行を一つ一つ確認していくことで行われるが、制御の安全性・信頼性という点からすると、それだけでは不十分である。例えばエンジンのアイドルスピード制御の場合、その目的からみて仕様の記述は1000RPM程度の低回転領域に限られてくる。仕様に従ったデバッグ作業では、低回転領域でのみ動作確認をすることになるが、実際のエンジンでは、6000RPMを越えることもある。このときアイドルスピード制御が作動し、スロットルを閉じるような動作を見逃したのでは確実なデバッグが行われたことにはならない。これは極端な例ではあるが、エンジン制御である以上、仕様に明記されていない条件下でもエンジンの作動する全領域において、制御動作の確認が必要なのは当然である。

しかしながら、仕様の全項目について全領域の動作を、一つ一つ確認していくには膨大な工数が必要となり現実的ではない。グループによる設計が行われている場合は、プログラム全体の動作把握が困難となり、予期せぬ不具合が発生する危険

性が更に高くなる。

2. 3 プログラムの検証（問題解決のために）

以上の問題点は、プログラムが大規模化し、プログラム全体の処理・動作を把握することが困難となったために起こるものである。これを解決するには、プログラムの内容を把握しやすいものとする必要がある。「ウォータフォールモデル」に基づくプログラムの設計手法は、上流から下流へと進めていくことを基本としている。実際には、各工程間でのフィードバックを何度となく繰り返して設計が行われており、このフィードバック回数を少なくし、あるいはサイクルタイムを短縮することが設計の効率化につながる。このためには、設計工程の下流から上流を検証できるシステムが必要となる。プログラムから、対象とする要求仕様に係わる処理内容・実行条件を表示することができれば、設計意図に反した動作が行われていないかを容易に検証でき、デバッグの初期段階で不具合を発見することができる。特に、フィードバックが頻繁なテスト工程（デバッグ）で有効である。

不完全な構造化によるモジュール・プログラミングの最大の問題点は、他のモジュールとの関連がつかみ難いことであるが、プログラムから逆に処理内容・実行条件を示すことができれば、設計者の担当したモジュールとの関連を含めて把握することが容易となる。結果として他のモジュール仕様の誤解や、連絡ミスといったことに起因する不具合を未然に防ぐことが出来る。

高級言語では、パソコンやワークステーションを利用した多くのプログラム開発ツールが発表されている。CASE（Computer Aided Software Engineering）と呼ばれるこれらのツールの多くは、設計の自動化やＣＡＤ化を図ったものである。ＰＡＤ図などのチャートから自動的にソースコードを生成したり、逆にソースコードからチャートを作成するものもある。しかしながら、アセンブラ言語では高級言語の場合と異なり、アセンブラの知識がなければチャートを見てもプログラムの内容を把握するのは難しい。大規模なプログラム開発では設計の効率化と共に、できあがったプログラムから逆に設計を検証する作業を効率化する必要がある。そのためには、プログラムの動作を難解なプログラム言語ではなく、もっと一般的な表現として表す必要がある。ＣＡＰＡＳはアセンブラで書かれたプログラムの実行条件、実行内容を日常使用している数式や表記方法で表し、プログラム動作を容易に把握できるよう開発したシステムであり、以上に述べたようなトップ・ダウン式のプログラム開発の欠点を補い、アセンブラ言語によるプログラム開発の効率化を行うためのシステムである。

3. ＣＡＰＡＳ概要

ＣＡＰＡＳはアセンブラ言語で記述されたプログラムを分かり易く表現することで、プログラム開発の効率化を狙ったシステムである。アセンブラ言語の場合、一つの命令語ではプログラム処理としての意味を持たない。幾つかの命令の組合せにより、はじめて意味をもつ。したがって単にフローチャートとして表してもアセンブラ言語のソースリストをそのまま読むのとさほど変わらない。ＣＡＰＡＳではプログラムの流れに沿っていくつかの関連する命令語をまとめ、数式や理論式など日常的な表現に置き換えて表している。具体的にはメモリに格納される内容や各分岐点での分岐条件をプログラムから解析し、フローチャートと併記している。一部日本語化を試みており、アセンブラ言語の知識がない人でもある程度のプログラム動作を知ることができる。

その他プログラムの解析結果を利用し、制御プログラムの設計確認に必要な検査機能をもたせている。アセンブラ言語の文法上のエラー（使用方法や書式の誤り）については、マシン語に翻訳する段階でアセンブラ自体に検出する機能をもっている。ＣＡＰＡＳはさらに検査機能を高めており、実行内容にまで立ち入ったプログラムミスをチェックする機能も持たせている。予め仕様条件の検査方法をプログラムとして登録しておけば、モジュールの使用方法のミスを検出することも可能である。この機能は過去に発生したバグの再発防止策として有効である。

3. 1　システム構成

プログラムは分岐命令に出会う毎に二つの流れを持つことになり、プログラムの解析を行うには単純に計算すると分岐命令数の二乗のプログラム処理が存在することになる。実際にはあり得ない条件の組合せや、条件によっては実行されない経路があり、もっと少ない数となるが膨大な量のデータを扱う必要がある。このため、出来るだけメモリ容量が大きく処理速度の速い計算機が必要となる。また、フローチャートを表示するため扱い易いＣＡＤ機能を持っていることが望ましい。

今回の開発では富士通製のワークステーションＧ250Ａを使用している。メインメモリ16ＭＢ、ＤＩＳＫ容量390ＭＢを装備し、出力装置にはＡ３のレーザプリンタを使用している。処理速度とフローチャートの作図にＰＩＣＡＤ（ユーザコマンド開発ツールを持ったＣＡＤソフト）が利用できることが主な選択理由である。

3. 2　基本原理

アセンブラはニーモニック・コードと呼ばれる記号を、ＣＰＵが実行可能な形であるマシン語に一語一語、置き換えていくものである。したがってニーモニックはＣＰＵの基本動作をあらわしている。制御用に特殊な命令体系を備えたものもあるが、ＣＰＵの基本動作は、演算・代入・分岐の三つに分類される。演算はプログラムの処理目的を実現するために行われるデータ加工であり、代入は演算に用いるデータ入力と演算結果のデータ出力を行う機能である。分岐は演算結果や入力データを判断し、次の演算やデータの入出力先を変更する機能で、プログラムを設計する上で最も重要なポイントとなる。データの出力は狭い意味でプログラムの処理目的であり、このデータの入出力と演算の繰り返しで最終的に目的とするデータが得られ、メモリや出力レジスタにデータが格納（ストア）される。このデータがプログラムの処理結果であり、最終的にデータが格納されるまでに通過した分岐命令の通過条件が実行条件となる。従って、メモリへのストア内容と、その実行条件が分かればプログラムの実行内容をつかむことができる。

解析は大きく分けて、①流れ構造解析、②分岐条件解析、③ストア内容解析、④フローチャートの４つに分類できる。

①流れ構造解析

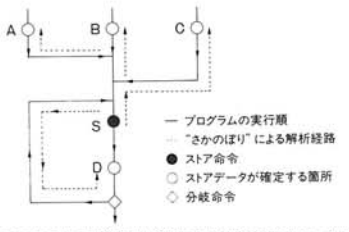
プログラムされた命令の実行経路を解析するソフトを流れ構造解析と呼んでいる。分岐命令に絶対アドレス分岐、相対アドレス分岐、インデックス分岐がある。単純な分岐はオブジェクトコードから容易に求まるが、インデックス分岐ではインデックスレジスタと呼ばれる演算レジスタの内容により分岐先が変わってくるため、インデックスレジスタに代入される値を求める必要がある。代入される値は後述のストア内容解析と同様な手法で求めることが可能であるが、この手法は流れ構造が解析されていることが前提となる。

現在のＣＡＰＡＳではどんなプログラムの組み方をされていても完全に分岐先を求めるということはできないが、実用上問題ないレベルと考える。職人芸のようなプログラミングが高い評価を得ていた時代と異なり、現在では分かり易い、合理的なプログラミングが必要とされているからである。

②ストア内容解析

命令の実行順に従って、実行される結果を一つ一つ追いながら実行内容を求めるには、命令の実行によるデータ変化やＣＰＵの状態変化を総て記憶しておく必要があり、膨大なメモリ容量が必要となる。また、プログラムでは処理の都合上最も便利な位置でデータを処理する為、必ずしもストア以前の命令でデータが確定しているとは限らず、実行内容が求まらない場合もある。

ＣＡＰＡＳでは、解析しようとするストア命令からプログラムの実行順とは逆の方向に、そのストア内容と関連する命令を探しながら解析を行っている。これを「さかのぼり解析」と呼んでいる。この方法だと、解析に必要な情報を漏れなく効率よく収集することが可能である。実際には取り得ない経路の存在やスタックの内容が変更されてサブルーチンのリターン先が変更されている場合の解析等、問題点が幾つかある。これらの問題に対しても正確にプログラム経路をたどれるよう流れ構造解析の段階で工夫をしている。図－4参照



Ｓを起点にプログラムの実行順とは逆方向に解析がすすむ。必要なデータのみを収集すればよいので、効率よく解析できる。

図－4　さかのぼりによるストア内容解析

Fig. 4　Store data analysis by back-trace method

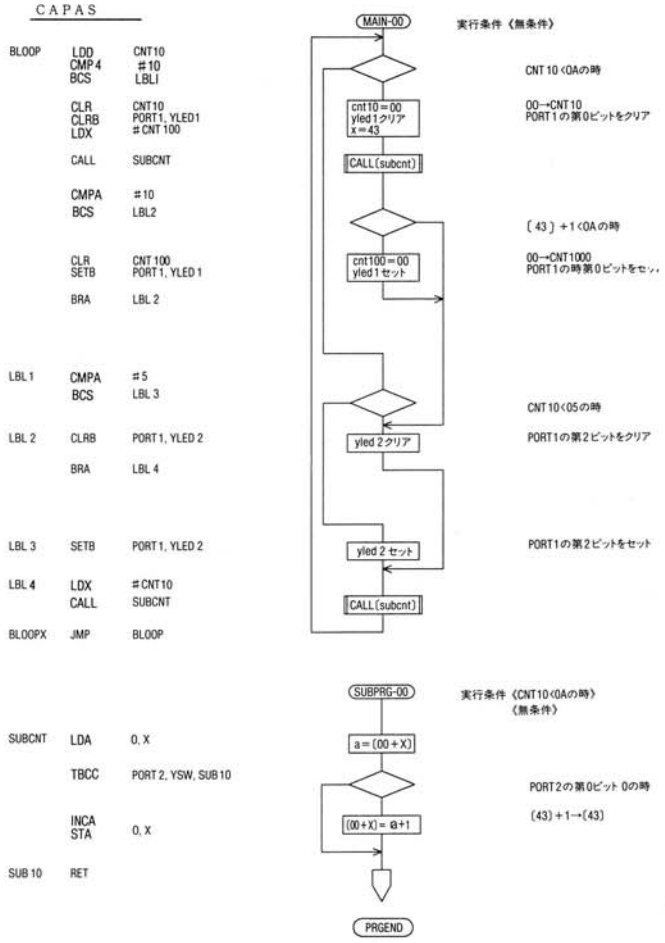
③分岐条件解析

分岐条件もストア内容解析と同様に「さかのぼり解析」を使用している。さかのぼりで重要なことは、求めようとする情報に関係する命令か否かを判断することと、必要な命令が総て求まったか否かを判断することである。詳細は避けるが、命令の実行によりＣＰＵのレジスタに与える影響を各命令毎にデータとして持っており、このデータを利用して要不要の判断を行っている。命令の合成は命令毎に実行内容を合成する関数を用意している。

④フローチャート

図－５はＣＡＰＡＳ解析結果の出力例である。単なるフローチャート・ジェネレータと異なり、ストア内容と分岐条件をフローチャートに併記している。これによりプログラムの流れや実行内容が比較的容易につかめる。アセンブルリストを眺めながらプログラムの動作を追っているのと比べると、格段に理解し易くなっている。チャート図の中に記述した内容は、プログラムの該当する箇所の命令を集め、数式や理論式としてまとめたものである。

各プログラムモジュール毎にフローチャートをまとめており、モジュールの実行条件を先頭に表示している。モジュールの実行条件は、そのモジュールに至る経路上に存在する分岐命令の通過条件（分岐しない条件）を論理代数演算（論理値を求めるのではなく、代数として表した分岐条件の論理式を簡略化することをいっている。）により簡略にまとめ直して表記している。モジュール・プログラミングにおいて、各モジュールが配置された位置の妥当性をみるのに役立つ。



図−5　出　力　例
Fig. 5　Output sample

4.　ＣＡＰＡＳの効果

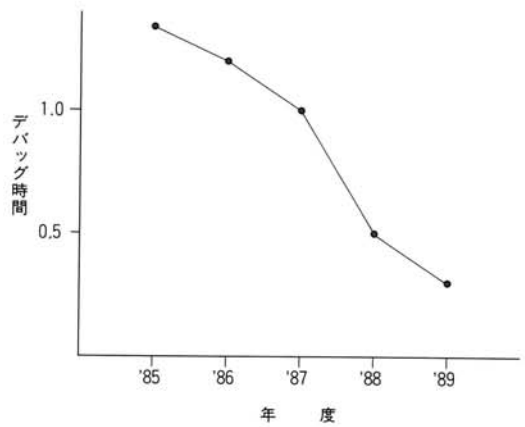
ＣＡＰＡＳの直接的な効果としては次の項目があげられる。

①フローチャートから、設計意図と反したところへ分岐している箇所を容易に発見できる。

②分岐条件総てを表示しており、仕様に反した条件下で動作していないかの判断ができる。

③ストア内容から、要求仕様に合わないデータがないか確認できる。

④プログラム検査機能により、不用意なバグの発生を未然に防ぐ。



図−6　デバック時間の推移
Fig. 6　Transition of debug time

⑤モジュールのブラックボックス化を防ぐ。

（設計ドキュメントの再製作が容易にできる）

図－6は当社におけるエンジン制御プログラムのデバッグ時間の推移を表したものである。モジュール・プログラミングとＣＡＰＡＳの利用が始まった年度より、デバッグ時間が確実に減少していることがわかる。

5. おわりに

自動車制御の分野においても徐々に高級言語が用いられ始めている。本格的に採用されるには、処理速度の問題とメモリ容量の問題が解決される必要があるが、やがて高級言語を用いて開発することが一般化すると考えられる。開発言語としてはＣ言語が有力である。ハードよりのプログラミングが可能なことや、生成されるオブジェクトサイズが小さいといった理由によるものである。ただし、設計時に生成されるオブジェクト（マシン語）を意識した設計をする必要があることや実行時間の把握が困難といった問題点がある。また、設計したプログラムのソースリストは、アセンブラほどでは無いにしても、非常に読み辛いものである。こうした理由から、高級言語の採用が直ちにソフトウェア開発の効率化につながるというものではない。これらの問題の谷間を埋める方策としてもＣＡＰＡＳの有用性がある。

現在、ＣＡＰＡＳはアセンブラ言語のみをサポートしているが、基本的な考え方は高級言語にも応用可能である。実行内容を分かり易くまとめることの他、条件別に実行時間を算出することも可能であり、高級言語の分野でもＣＡＰＡＳは有効なツールとなりえる。ＣＡＰＡＳではＣＰＵの出力信号に関する実行内容を頂点に、関連する変数を求めた実行内容をツリー状に配置することでプログラム全体での実行内容を示そうと試みている。実行条件が厳密すぎることや、実行内容が機械的すぎるなどの理由から実用的なものになってはいないが、プログラム設計の信頼性を確保するにはプログラムから逆に仕様を導きだすツールとすることが理想である。

（参考文献）

- 小島、稲葉：“ソフトウェア開発の新手法、プロセス・プログラミング”，日経エレクトロニクス，1987. 5. 18（No.421）
- 稲葉：“ソフトウェア要求分析用ツールの商品化が進む”，日経エレクトロニクス，1987. 6. 1（No.422）
- 高野・水野・坂下・北畠：“分散型開発システムで仕様書作成からプログラム生成までを支援する”，日経エレクトロニクス，1987. 7. 13（No.425）
- 前澤・葉木・津田・印東：“ソフトウェア生産技術の展望”，日立評論，1988. 2（vol.70）
- 萩谷：“ソフトウェア考現学”，ＣＱ出版（1985）
- 淵・黒川：“新世代プログラミング”，共立出版（1986）
- 二村：“プログラム技法”，オーム社（1984）