ソフトウェア工学 129－4
（2000．11．16）

プログラムスライシングを応用した業務アプリケーションからの情報抽出

松尾昭彦*、長橋賢児*、直田繁樹*、平田正一**、須藤茂雄***
*富士通研究所 **富士通 ***富士通青森システムエンジニアリング

アプリケーションの再構築の際に、旧システムの資産を再利用することができれば工数の削減につながる。そこでファイルレコードの更新処理を行うプログラムを対象に、その仕様情報を抽出する方法を提案する。この方法はプログラムスライシング技術を応用して CUD(チェック更新定義書)の形式で情報を抽出する。ファイル出力文を基準にした逆方向スライシングの結果から、ファイルレコードの更新処理とその実行条件を求めることができる。この方法は実際に再構築を行っている業務アプリケーションで試行を行い、有用な出力結果が得られたとの評価を得た。

A specification extraction method from a business application using program slicing

Akihiko Matsuo*, Kenji Nagahashi*, Shigeki Suguta*, Masakazu Hirata**, Sigeo Suto***
*Fujitsu Laboratories **Fujitsu ***Fujitsu Aomori System Engineering

While rebuilding an application system, it will reduce the cost to reuse the old system's information. In this paper, we propose a specification extraction method from a business application which updates file records. The method uses program slicing technique and extracts information in CUD(Check Update Description) style. Using the results of backward slicing from file output statements, record update processes and their executing condition are able to be extracted. The method was applied to a project which was rebuilding a business application, and was evaluated that the extracted CUD contained useful information.

1. はじめに

近年、業務アプリケーションを開発するにあたっては、まったくの新規開発というケースは少なくなり、旧システムが存在する上で新システムを開発する場合がほとんどである。例えば、これまで専用端末から入力される伝票をバッチ処理で処理していたシステムを、Web 端末を使い対話的に処理するシステムとして再構築するようなケースが増えている[11]。このような場合、追加機能や新しい言語・動作環境への対応などが新システムの設計の中心であり、業務ロジック部分はほとんどそのまま旧システムのものを流用できることになる。また、企業の合併を機会に新システムの導入を図るようなケースもあり、このような場合も含め、旧システムの機能を継承しつつ短期間での開発が求められる場合が増えている[9][10]。

ソフトウェア開発の生産性と質を上げる最上の方法は新たに作る量を減らし、試験済み、検証済みのコードを再利用することである[8]。このため、旧システムの仕様から変化がない部分について、何らかの形で旧システムの資産を再利用することが望ましい。このうち、データの再利用については情報抽出が比較的容易であり、これまでの開発でもよく行わ

れてきたが、プログラムや仕様情報の再利用は困難であった。そこで我々は、言語や動作環境が変わっても流用が可能なプログラムの仕様情報に注目し、旧プログラムからその業務ロジックを表す仕様情報を抽出する方法の研究を行っている。

以下、2章では既存システムの再利用の形態について概観する。3章では業務ロジックを表現する仕様ドキュメントの一つであるCUDについて説明し、プログラムスライシングを応用してこれを生成する方法について提案する。そして4章では再構築を行っているアプリケーションを対象とした試行の結果とその評価について述べる。

2. 既存システムの再利用の形態

旧システムの資産を再利用する場合、その形態としては、データの再利用、プログラムの再利用、仕様情報の再利用の3つが考えられる。このうちデータについては情報抽出が比較的容易であり、これまでの開発作業においてもしばしば行われてきたが、プログラムや仕様情報の再利用は困難であった。

2.1 データの再利用

2.1.1 レコードレイアウト情報の再利用

旧システムが使用しているデータベースやファイルをそのまま使用したり、新たな要求に対応して改良を加えるようなケースでは、旧システムのレコードレイアウト情報が得られれば設計が容易になる。レコードレイアウト情報はデータベースのスキーマ定義という形でプログラムから分離していたり、あるいはプログラムのデータ項目定義部分だけを調査することで比較的容易に情報が得られるため、この情報の再利用はこれまでの開発作業でもよく行われている。レイアウト図の自動生成などのツールも多い。

2.1.2 ドメイン情報の再利用

旧システムが使用している各データの属するドメイン(意味)や他のデータとの関係は、仕様情報がないと把握しずらく、名前やレイアウト情報だけによる分析では重要な関係を見落としてしまうことがある。この問題に関しては、以前我々はプログラム中の代入などによる関係を分析して自動的に変数の分類を行う方法を提案した[7]。この技術を応用することでデータの属するドメインを明らかにしたり関連のあるデータを見つけ出すことができる。また、それらのデータと比較されている定数の情報を検索することで、コード定義などデータのドメインに付随する情報を取得することが可能である。

2.2 プログラムの再利用

2.2.1 ラッピングによる再利用

既存システムをそのまま利用し、外部にインターフェースとなる層を被せて新システムと結合するような再利用方法が考えられる。この方法は、再構築後にも旧システムがそのまま運用されることになるため、例えばWebクライアントへの対応など、システムへの要求に大きな変化がない機能追加の場合には有効であり、既に品質が確保されている旧システムをそのまま流用できるため開発工数も大幅に削減できるという魅力がある。その一方で、バッチ処理をリアルタイム処理に変更するなど全体の構成や運用に大きな変化がある場合にはこの方法によって旧システムを再利用するのはほぼ不可能である。

2.2.2 プログラムから抽出した部品の再利用

旧システムの仕様がそのまま流用できる部分のプログラムソースを再利用可能な形で部品化し、新システムに取り入れて使うことができれば設計・実行工程での工数の削減に大きく貢献できると思われ、このための研究も行われている[9]。新システムでの実装言語が異なる場合に、そのまま使える部品を直接取り出すのは困難であるが、詳細設計・実装時に参照する仕様情報の一つとしての有効性は期待できる。

しかし、現実には一般に業務で使われるシステムのプログラム数は膨大であるため、人間が部品化すべき部分や範囲を指定するのは難しいという問題がある。また、部品を再利用するためには仕様に関する情報が必要であるが、抽出した部品の仕様を人間が理解しやすい形で表記するのもまた困難である。

2.3 仕様情報の再利用

2.3.1 文書化された仕様情報の再利用

旧システムの設計・実装時に使用した文書化された設計情報を再利用することは、再構築の際にしばしば行われている。ただしこの場合、仕様書が正しくメンテナンスされており現在のシステムの内容を表していることが前提となるが、多くの場合、プログラムだけが修正・機能追加され仕様書の内容と一致しなくなっているので、プログラムソースも確認する必要が出てくる。また、仕様情報のフォーマットはシステムによってまちまちで電子化されていない場合も多い。このため、内容を機械的に処理して再利用できるのはごく限られたケースに限られ、その手法も特定の書式に依存したものになりがちである。

2.3.2 プログラムから抽出した仕様情報の再利用

旧システムのプログラムソースから、プログラムソースよりも理解が容易な形式で仕様情報を抽出することができれば新システムの設計・実装工程の工数を削減することが可能である。一般的なプログラムに対する仕様情報の抽出は困難だが、業務プログラム固有の処理形式に注目した上での仕様抽出の方法についてはいくつか提案されている[4][5]。また、抽出した情報をプログラム自動生成に利用することで、言語や動作環境によらない再利用も考えられる[10]。

3. プログラムからの仕様情報の抽出

我々は、COBOLで記述された、レコードの更新処理が中心となるバッチ処理の業務アプリケーションプログラムを対象に、プログラムソースから仕様情報をCUD形式で抽出する方法を検討した。対象としてCOBOLアプリケーションを選んだのは、現在の業務アプリケーションの多くがCOBOLで記述されており今後再構築の対象となっていくことが予想されるためである。またこの形態の再利用が有用であると判断したのは、プログラムそのものの再利用では適用可能な状況が限られており、仕様書からの情報抽出はその形式に依存してしまうためプロジェクト毎に異なる手法がが必要になって汎用性に乏しいためである。プログラムから仕様情報を抽出することができれば、言語や動作環境によらず、多くの再構築プロジェクトにおいて新システムの分析段階から情報を活用することができる。

3.1 CUD形式での業務ロジック情報の抽出

ファイルやデータベースのレコードの更新処理が中心となる業務アプリケーションの仕様情報の表現には、CUD(チェック更新定義書)を用いるのが有用である。CUDとは、業務ロジックを表す仕様情報の一つであり、データ中心開発技法に基づく統合CASE AA/BRMODELLERがプログラム自動生成に使用する業務仕様書の一つである[5]。CUDは{変数・代入値・実行条件}の3つ組でファイルレコードの更新を表現する更新記述と、{エラー処理・エラー条件}の組でエラーチェックを表現するチェック記述を中心とする表形式の仕様書である。この形式を用いれば、プログラムの知識がなくても業務ロジックの仕様が理解できるためプログラマではない専門家でもロジックの記述が可能になるため、有用な設計ドキュメントの一つと言える。図1にCUDの例を示す。

そこで我々は、レコードの更新処理が中心となるバッチ処理の業務アプリケーションプログラムを対象に、プログラムソースから仕様情報をCUD形式で抽出する方法を検討した。対象となるプログラムの性質をレコードの更新処理に限定することで、抽出すべき業務ロジックの位置が明らかにでき、大量のプログラムの中から分析対象となるコードを自動的に

項目仕様名	識別名
振込データ編集処理	口座振込レコード

データ項目	?/=	処理	条件
口座番号	?		口座検索(口座番号[口座振込入力電文]) = 0
金融機関名	=	金融機関名.口座検索(振込先口座[口座振込入力電文])	
支店名	=	支店名.口座検索(振込先口座[口座振込入力電文])	
普通当座種別	=	"普通"	口座区分[口座振込入力電文] = 1
普通当座種別	=	"当座"	口座区分[口座振込入力電文] = 2
口座番号	=	口座番号[口座振込入力電文]	
預金者名	=	預金者名.口座検索(振込先口座[口座振込入力電文])	
振込金額	=	振込金額[口座振込入力電文]	

図1 CUDの例

見つけ出すことができる。

3.2 スライシングを応用したCUDの抽出

プログラムソースからCUDの抽出を行うため、我々はプログラムスライシング技術[1]を応用した更新仕様情報の抽出方法を提案する。

CUD 形式での仕様情報の抽出については以前にも、プログラム内の一連の処理をブロックとしてCUD化し、それをつないでいくことでプログラム全体を表すCUDを作成する方法について提案を行っている[5]。しかしこの方法では、生成されるCUDは処理の流れに従ったものになるため、抽出できるCUDの形はプログラムの書き方に大きく依存してしまうという問題があった。それに加えて、プログラムの制御構造によってCUDが分断され大きくすることができないという問題や、処理ブロックにある代入文がレコードの項目に対してどのような関係にあるのかが明確でなく、プログラム中からレコードの更新に関する情報をうまく取り出すことができないという問題があった。

そこで我々は、処理の並びの順にCUDを生成していくのではなくて、データをファイルに出力する箇所に注目し、その箇所におけるレコードの値をプログラムの実行経路を遡って求めていくことでCUDを生成する方法を提案する。この方法を使うことで、処理の並び順や制御構造を気にすることなく、注目するレコードの値を更新する処理を見つけ出してCUDにすることが可能である。

プログラムの実行経路を遡って値を更新している箇所を求めることは、プログラムスライシング技術によって実現することができる。プログラムスライシング技術はプログラム内の命令の間の依存関係を明らかにする技術であり、テスト、デバッグ、保守、メトリックスなど、広範囲に応用されている有用な技術である[2]。

レコードの値をプログラムの実行経路を遡って求めるには、ファイル出力文とレコード項目をスライシング基準として逆方向スライシングを行えばよいことになる。

3.3 STEM構造による実行経路情報の表現

スライシングの実現のため、我々はCOBOLの特性を考慮した階層的な制御構造モデルを用意した。COBOLはGO TOなどの非構造化命令や、PERFORMなどのサブルーチンコール命令があり、プログラムスライシングのためには特別な配慮が必要である。このため、COBOLの特色を考慮したスライス計算方式が提案されているが[3]、我々は、STEMと呼ぶ階層的な制御構造モデル[5]で実行経路を表現した上でスライシングを行う方法を採用した[6]。プログラム中のGO TO命令などは、STEM構造の上ではジャンプを含まない構造化された形式で表現される。このモデルを用いることで、スライシングのように実行順序を考慮する必要のある処理や、文の実行条件を取得する処

```
SORT-MAKE SECTION.
  IF DENPYO-ID = '05' THEN
    MOVE SHITEN-CD TO SORT-CD
    IF SHORI-CD = '92' THEN
      MOVE SPACE TO SORT-SHORI
      GO TO EXT
    ELSE
    END-IF
  END-IF.
  PERFORM SORT-REC-MAKE.
EXT.
  EXIT.
```

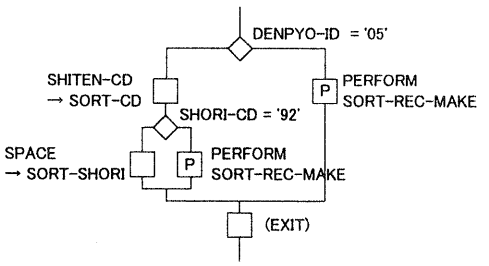

図2 STEM構造の例

理が極めて容易に実現できる。図2にプログラムの実行経路をSTEM構造で表現した例を示す。

STEM構造上では、個々の文は一つのノードとして表現される。文の持つ代入効果はそのノードの代入元、代入先の組として表現される。ファイル入出力文は代入文の一種として扱う。

条件分岐文は内部にそれぞれの分岐経路を表すSTEM構造を持ったノードとして表現される。PERFORMによるサブルーチン呼び出しやループ構造は、内部に処理を表すSTEM構造を持ったノードとして表現される。

図2の例は、GO TOを含むプログラムを構造化し、元の実行経路と等価でジャンプのない形に変形している所を示している。ただしこの変形によって、元は一箇所にしかなかったPERFORM文が構造中の二箇所に現れているため、この形のままスライシングを行うとPERFORM文の部分が二回分析対象となってしまうという問題が生じる。

このような複数箇所への出現が発生するのは、分岐処理の一方がGO TOなどにより構造を崩した飛び出しを行って別の経路に合流しているためである。このため、合流を表現する構造を導入することができれば複数箇所への出現を抑止することができる。この構造としては、C++やJavaなどの言語で使われる例外処理（THROW-CATCH）のメカニズムが考えられ、これを追加したSTEM構造を用いることで複数箇所への出現の問題は回避できる。図3に、THROW-CATCHを表す要素を含んだSTEMによって表現したプログラムの構造を示す。

なお、STEM構造は再帰的な呼び出しがあった場合に無限循環構造になってしまうが、COBOLでは再帰的な呼び出しが仕様上禁止されているため、正しいプログラムであれば分析上の問題とはならない。

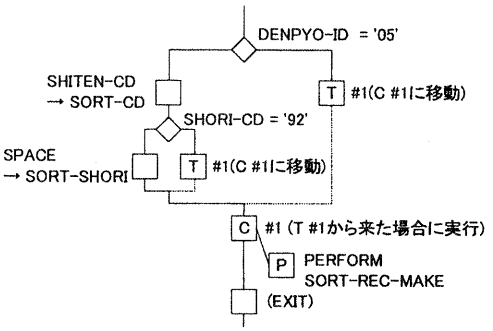

図3 THROW-CATCH構造を導入した例

3.4 STEM構造上でのスライシングの実現

STEMはプログラムを有向グラフで表現したモデルであり、この上でデータフローを追跡することでスライシングを実現することができる。また、すでに構造化されているため、データフローを追跡する処理はきわめて単純になる。

順方向のスライシングは、スライシング基準となる文に対応するノードを出発点として実行順にノードを辿っていき、そのノードがスライシング対象の変数を代入元として使っているなら、その文をスライスして代入先の変数をスライシング対象の変数に追加する。そのノードがスライシング対象の変数の値を代入先として使っているなら、代入先の変数をスライシング対象から除外する。この手順をプログラム末端のノードに到達するまで繰り返すことで実現

できる。分岐構造がある場合は、それぞれの分岐構造のスライシングを行い、その結果をマージする。

また、逆方向のスライシングは実行順の逆にノードを辿っていき、そのノードがスライシング対象の変数を代入先として使っているなら、その文をスライスして代入元の変数をスライシング対象に追加し、元の変数を除外する。この手順をプログラム先頭のノードに到達するまで繰り返すことで実現できる。

PERFORM によるサブルーチン呼び出しに関しては、引数の受け渡しは発生しないため、単純に呼び出し先となる STEM 構造をスライシングするだけでよい。CALL による別プログラムの呼び出しに関しては引数の受け渡しがあるため、サブルーチン側のスライシング基準を用意し変数を引き継いだ上でサブルーチンの STEM 構造をスライスするなどの考慮が必要になる。

ループ構造がある場合、ループ内の STEM を一度スライシングしただけでは正確な結果が得られないことがある。正確なスライシングを求めるためには、ループ内の分析は複数回行い、分析終了時のスライシング対象の変数に新たなものが追加されなくなるまで繰り返せばよい。

3.5 更新定義の抽出方法

更新定義の抽出のためには、レコードを対象となるファイルに出力している箇所を見つけて逆方向スライシングを実行すればよい。これは通常はファイル出力文そのものになるが、アクセスルーチンを経由している場合もあり、そのような場合はアクセスルーチンの呼び出し箇所に注目するほうが分析量が減るため処理効率がよい。

スライシングの結果として得られたノードの代入元、代入先、実行条件を表形式で出力することで更新定義を作成することができる。実行条件については、ノードの祖先となる分岐ノードを列挙し、その分岐ノードが持つ条件を AND 結合していくことで得られる。

3.6 チェック定義の抽出方法

チェック定義の抽出のためには、プログラム中でエラー処理を行っている文に注目し、その文の実行条件を求めればよい。エラー処理を行っている文としては、特定のセクションへ制御を移している文、あるいは状態を示す変数にエラーを示す値を代入している文などが考えられる。これらのエラー処理文の識別には外部からの情報が必要ではあるものの、対象となる文の実行条件を求める処理は更新条件同様、STEM 構造上で容易に実現できる。

4. 評価と問題点

我々は今回提案した手法を再構築中のアプリケーションに適用し、その旧システムを分析してプログラムスライシングを応用した更新定義抽出を行い、仕様情報としての有効性の評価を行った。なお、チェック定義の抽出についてはエラー処理の指定が必要になるため、今回の試行では行わなかった。

4.1 対象アプリケーションの規模

試行を行ったアプリケーションは COBOL で記述された物流管理のシステムで、プログラム 398 本、合計で 312k ステップの規模であった。これらのプログラムは構文木情報と STEM 構造を解析情報リポジトリに格納しておき、更新処理抽出はそのリポジトリを参照しながら処理を行った。解析環境は WindowsNT 4.0 上の PC で、一連の処理に要した時間は数時間程度であった。

4.2 試行結果と評価

図 4 に、この試行によって得られた更新定義の一部を示す。出力結果は、更新処理の内容がレコード項目名、代入値、実行条件の 3 つ組で表現された表の形式になっており、プログラムの構造を追わなくても処理内容が把握できる。出力の一部には後述する

レコード名	レ	データ項目名	=	処理	条件1	条件2	条件3
出力REC	4	出力-売上金額					
出力REC	1	出力REC	=	入力REC	入力-店コード[入力REC] = SPACE【(155)】		
出力REC	4	出力-売上金額	=	売上金額	入力-品コード[入力REC] = "1000"【(163)】	NOT (入力-店コード[入力REC] = SPACE)【(155)】	
出力REC	4	出力-売上金額	=	売上金額	計算区分 = SPACE【(168)】	NOT (入力-品コード[入力REC] = "1000")【(163)】	NOT (入力-店コード[入力REC] = SPACE)【(155)】
出力REC	4	出力-売上金額	=	売上金額	NOT (計算区分 = SPACE)【(168)】		
WORK	3	売上金額	=	包装料	入力-品コード[入力REC] = "1000"【(143)】	入力-店コード[入力REC] NOT= SPACE【(640)】	
WORK	3	売上金額	=	(通用商品単価 * 入力-数量[入力REC] / 12) + 包装料 + 荷造料	計算区分 = ZERO OR SPACE【(85)】	NOT (入力-品コード[入力REC] = "1000")【(143)】	入力-店コード[入力REC] NOT= SPACE【(640)】
WORK	3	売上金額	=	WK-商品単価 * 入力-数量[入力REC] / 12	計算区分 = "1"【(122)】	NOT (計算区分 = ZERO OR SPACE)【(85)】	NOT (入力-品コード[入力REC] = "1000")【(143)】
WORK	3	売上金額	=	WK-商品単価 * 入力-個数[入力REC]	NOT (計算区分 = "1")【(122)】		
WORK	3	通用商品単価	=	WK-商品単価 * 割引率テーブル(1)	入力-顧客区分[入力REC] = 1【(825)】		
WORK	3	通用商品単価	=	WK-商品単価 * 割引率テーブル(2)	入力-顧客区分[入力REC] = 2【(830)】		
WORK	3	WK-商品単価	=	入力-税込単価 [入力REC]	入力-種類コード[入力REC](1 : 1) = "9" OR (入力-種類コード[入力REC](1 : 3) >= "010" AND 入力-種類コード[入力REC](1 : 3) <= "990") 【(1386)】	入力-区分[入力REC] = "1" OR "7" OR "8"【(67)】	
WORK	3	WK-商品単価	=	入力-商品単価 [入力REC]	NOT (入力-区分[入力REC] = "1" OR "7" OR "8")【(67)】		

図4 業務アプリケーションから抽出した更新定義

問題が顕著に現れたものもあったが、項目名の日本語化などの手法を併用することで可読性の向上が可能である。試行の結果は再構築の担当者により有用と評価され、旧システムの仕様情報として利用されることになった。

この更新定義の記述内容はまだプログラムに近いものであり、本来CUDに求められるレベルでの記述が直接得られるわけではないが、設計・実装作業のベースとしてこれを使うことで作業工数の削減を図ることができる。また、以前提案した抽出方法と比較して、プログラムの実装に関わらずレコード更新に関わる処理をデータ別に並べることができるのでプログラムの制御構造を追わなくても処理内容を理解することができる。さらに、スライシング結果として得られる代入文をそのまま取り出せばプログラムソースレベルでの再利用をすることも可能である。

4.3 問題点

問題としては、実行条件や代入値がプログラム中の記述のまま入ってしまうため、理解が難しい場合があることが挙げられた。特に条件については、例えばサブルーチンからのリターンコードをチェックして処理をするような場合、どの箇所でも同じ作業変数を比較しているように見えるが、それぞれの箇所で違う意味の値が入っていることになり、単純に条件式に書かれた内容を出力するだけでは正しい意味を汲み取ることができない。代入値も中間変数を使っている場合や記号的な変数名が用いられている場合、その意味を読み取ることが難しいことがある。

これらの問題の改善のためには、条件文や代入値をそのまま出力するのではなく、それらが持つ意味を表す表現に変換して出力することが必要である。意味情報を自動的に求めるのは困難であるが、旧システムの仕様情報として変数名の日本語名標辞書やコード定義が用意されている場合もあるため、これらを利用する方法が考えられる。

また、中間変数の問題については、その値の代入元となる定数や変数を検索し置き換えることで表示を判りやすくすることが考えられる。代入元の値はデータフローを調べれば判るので、この置き換えはSTEM構造上で容易に実現することができる。

なお、複数の条件がネストして一見複雑なものになっているケースでも、よく条件式の内容を見てみると論理的に単純化が可能な場合がある。このような場合、条件式の内容を論理演算によって簡略化することが考えられる。図5に、簡略化が可能な実行条件の例を示す。この場合、「当座」の値に対する実行条件はSYORI-CD = '2'に簡略化できる。

```
IF SHORI-CD = '1' THEN
   MOVE '普通' TO WK-SYUBETSU
ELSE IF SHORI-CD = '2' THEN
   MOVE '当座' TO WK-SYUBETSU
END-IF.
```

データ項目	処理	条件
WK-SHUBETSU	NC'普通'	SYORI-CD = '1'
WK-SHUBETSU	NC'当座'	not (SYORI-CD = '1') and (SYORI-CD = '2')

図5 実行条件の簡略化ができる例

5. おわりに

プログラムスライシング技術を応用することで、業務アプリケーションからCUD形式の仕様情報を抽出する方法についての提案を行い、その評価を行った。この方法を用いれば設計ドキュメントとして使用されているものに近い形式で仕様情報を抽出できるため、プログラムの仕様理解にとどまらず、新システムの設計工程で抽出した仕様情報を活用することも期待できる。今回の方法によって抽出した情報は記述内容のレベルがプログラムソースに近く、より理解しやすい高度な表現を取り入れて理解性の向上を図る必要はあるものの、実際に再構築を行っているアプリケーションを対象にした試行で有用であるとの評価を得た。

参考文献

[1] Weeiser,M., "Program Slicing", IEEE Transactions on SoftwareEngineering, PP.352-357, Vol. SE-10, No.4, July, 1984

[2] 下村隆夫、「プログラムスライシング技術と応用」、共立出版、1995

[3] 友納正裕、大竹和雄、小泉昌紀、川崎洋治、中島震、「COBOLを対象としたプログラムスライス計算方式」、情報処理学会ソフトウェア工学研究報告、SE-102-3, 1995

[4] 原田実、吉川彰一、永井英一郎、「COBOL プログラムから非手続き仕様を逆生成するリバースエンジニア CORE/M」、情報処理学会論文誌, Vol. 36, No.3, PP714-727, 1995

[5] 長橋賢児、上原三八、「アプリケーション再構築のためのリバースエンジニアリング技術」、第51回情報処理学会全国大会予稿集 5-193, 1995

[6] A. Matsuo, and S. Uehara, and M. Kimura, "A Maintenance SupportSystem based on High-level Control-Flow and Data Dependency", PP390-398, Proceedings of APSEC95, 1995

[7] K. Kawabe, and A. Matsuo, and S. Uehara, and A. Ogawa, "VariableClassification Technique for Software Maintenance and Application toThe Year 2000 Problem", CSMR98, 1998

[8] 竹下亨、「ソフトウェアの保守・再開発と再利用」、共立出版、1992

[9] 丸山勝久、島健一、高橋直久、「区間限定スライスを用いた部品生成システムの評価」、情報処理学会ソフトウェア工学研究報告、SE-105-7, 1995

[10] 佃軍治、団野博文、永岡郁代、「既存ソフトウェアの再利用におけるオブジェクト指向開発の支援環境の構築」、情報処理学会ソフトウェア工学研究報告、SE-111-3, 1996

[11] 方学芬、玉井哲雄、「再構築のためのソフトウェア解析アプローチ」、情報処理学会ソフトウェア工学研究報告、SE-127-4, 2000