バッチ処理システムにおけるデータセット間の依存関係の抽出

竹之内 啓太[1,a)]　石尾 隆[1,b)]　岡田 譲二[2,1,c)]　坂田 祐司[2,d)]　井上 克郎[1,e)]

概要：バッチ処理とは計算機システムに蓄積されたデータを一定期間ごとに一括で計算する処理の方式である．レガシーシステムには数多くのバッチ処理が含まれているが，いずれも 1 つ以上の入力データセット（データの系列）から 1 つ以上の出力データセットを作成する処理とみなすことができる．本研究では，バッチ処理システムが実行する個別のバッチ処理の概要として，各出力データセットが依存する入力データセットの集合を抽出する手法を実現した．具体的には，ジョブ制御言語で記述されたバッチ処理の上でのデータ依存関係解析に加え，バッチ処理の各ステップに対応する COBOL プログラム内部のデータ依存関係を解析することで，より詳細なデータ依存関係の解析を可能とした．ケーススタディとして，あるレガシーシステムのバッチ処理の分析に適用したところ，手法のデータ依存関係解析は多くのバッチに対し有効であり，COBOL プログラムの内部を考慮した詳細化が有効であるバッチが存在することが確認できた．

Extraction of Dependence Relation among Data-sets in Batch Processing System

Keita Takenouhi[1,a)]　Takashi Ishio[1,b)]　Joji Okada[2,1,c)]　Yuji Sakata[2,d)]　Katsuro Inoue[1,e)]

Abstract: Batch processing is a data processing mode which runs at regular intervals and treats a series of accumulated data. Legacy systems are mainly composed of batch processes and a batch process generates output data-sets using input data-sets. In this paper, we propose a method to extract output data-sets and their corresponding input data-sets as a summary of a batch process. This approach performs data dependence analysis for not only job control language but also COBOL programs to realize a more detailed analysis. A case study with a real legacy batch system shows that data dependence analysis on data-sets is effective in many batch processes and that the detailed analysis on COBOL is effective for a particular set of batch processes.

1. はじめに

バッチ処理システムとは，一定期間ごとに大量のデータを一括に計算するバッチ処理を実行するためのシステムである．企業活動を支えるソフトウェアシステムの中でも，業務における売り上げの集計や給与の計算といった基幹業務を果たすものであり，その多くは数十年前に開発されてから現在まで保守され続け，社会的に大きな役割を担っている [1], [2].

バッチ処理システムは，レガシーシステムと呼ばれているものの 1 つである．メインフレームコンピュータ上で稼働しており，システムとして社会的に大きな役割を担う一方で，長年の保守開発によりシステムの大規模化や複雑化が進んでいる傾向がある．さらに，ドキュメントが消失していることによりシステムの仕様が不明確であったり，COBOL やメインフレームコンピュータ等の古くなった技術に精通している技術者が現場から離れている問題もあり，

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[2] 株式会社 NTT データ
NTT DATA Corporation
[a)] t-keita@ist.osaka-u.ac.jp
[b)] ishio@ist.osaka-u.ac.jp
[c)] okadaju@nttdata.co.jp
[d)] sakatayu@nttdata.co.jp
[e)] inoue@ist.osaka-u.ac.jp

結果として運用・保守に必要な費用が増加している [3].

レガシーシステムを最新の技術を用いたシステムへと作り替えることで，さらに継続的な運用，保守を可能とする取り組みがレガシーモダナイゼーションである．過去になされてきたレガシーモダナイゼーションの研究としては，プログラムの自動変換，すなわち古いプログラミング言語で記述されたソースコードを比較的新しいプログラミング言語のソースコードに自動的に変換する取り組み [4], [5] が挙げられる．しかしながら，プログラミング言語の自動変換は言語レベルでのモダナイゼーションを行うものであり，システムの設計そのものを作り直すことにはならない．そのため，元のレガシーシステムの設計やデータ構造の悪い部分も新システムに引き継がれることになり，結果としてモダナイゼーションのレビューやテストの工程において，新システムにおける保守性の問題や実行速度などのパフォーマンス面での問題が生じることがある．これがシステム設計の見直しなど開発工程の後戻りを引き起こし，新システムのリリースまでに膨大なコストがかかる原因の 1 つとなっている．

我々の研究グループは，レガシーシステムの動作をそのまま新システムへと反映するのではなく，レガシーシステムから一度システムの動作の概要を抽出し，得られた情報に基づいてシステムの設計そのものを見直すことでレガシーモダナイゼーションを達成することを目指している．たとえば，COBOL プログラムの多くは業務データの集計処理を担当しているため，使用しているデータの入出力の関係や操作の意味を把握することができれば，リレーショナルデータベースの操作（SQL 文）として効率よく同一の意味を持つ処理を再実装するといった活動を行うことが可能となる．

本研究では，バッチシステムの動作の概要を抽出する技術の 1 つとして，1 つのバッチ処理を構成するプログラム群から，システム外部との入出力となるデータセット (バッチシステムにおけるデータの系列) 群を抽出し，どの入力データセットの値を使ってどの出力データセットの値を計算しているかのデータフロー関係を表形式で可視化する手法を構築した．具体的なアプローチとしては，ジョブ制御言語の記述から COBOL プログラムの実行順序と各プログラムにおいて使用するデータセット名称を抽出し，ジョブ制御言語上でのデータ依存グラフを構築する方法を定義した．また，複雑化したひとつの COBOL プログラムが互いに無関係な複数の処理を実装している場合に対応するため，COBOL のデータ構造に対応したプログラムスライシング手法を導入し，ジョブ制御言語上でのデータ依存関係のより正確な計算を可能とした．

提案手法の有効性を調査するために，ケーススタディとして，ある企業で利用されているバッチ処理システムのすべてのバッチに対して手法を適用した．その結果から，データ依存関係グラフを構築することによるデータセットの依存関係の解析が多くの場合で有効であることや，COBOL プログラムのスライシングによる解析の詳細化が有効である場合が存在することを確認した．また，手法の出力となるデータセットが具体的にどのくらいの規模になるのかを調査し，動作の概要を理解するために重要でないと考えられるデータセットを除外するという，手法での工夫点が有効に働くときがあることを確認した．

以降，2 章では関連研究について述べ，3 章で提案手法を説明する．4 章でケーススタディについて，5 章は妥当性への脅威についてそれぞれ述べる．6 章でまとめと今後の課題を述べる．

2. 関連研究

Hasselbring ら [6] は，レガシーシステムから多層アーキテクチャをもつ新システムへの移行の手法として Dublo パターンを提案している．この手法は既存のレガシーシステムのコードをそのまま活用するため，新たなユーザインターフェースと，ビジネスロジックを追加するための中間層を作成し，それらから間接的にレガシーシステムに接続するというものである．既存のコードやデータベース設計などをそのまま再利用できるということや，徐々に新システムへ機能を移行できるという利点を持つ一方で，機能に対応するコード片を事前にレガシーシステムから識別する必要がある．

Prez-Castillo ら [7] は，レガシーシステムに対するリバースエンジニアリングにおいて，コード片とアクセスされるデータベースの関係を抽出する手法を提案している．レガシーシステムのモダナイゼーションを行う際，元のすべてのコードが新システムにおいて使用されるとは限らない．一部のコードのみが新システムに移行されるとき，手法によりソースコードの移行と併せて必要なデータベースの移行を行うことが可能となる．手法はアーキテクチャ駆動モダナイゼーションと呼ばれるフレームワークに基づいており，ソースコードやデータベースなどはすべてモデル化されたのち，分析される．

Bao ら [8] は，レガシーシステムのなかでも比較的新しいオブジェクト指向言語によって書かれたシステムのモダナイゼーションのための，機能単位の抽出手法を提案している．この手法では，ソースコードのほかに仕様書を用いる必要がある．ユースケースから機能ごとのテストケースを生成し，そのテストケースの実行時のログを記録する．その後，ログに含まれるクラスやメソッドを解析することで，機能に対応するコード片の抽出を行うという，動的な手法に基づくものである．

Wiggerts ら [9] は，レガシーな COBOL プログラムから Object-Oriented COBOL へ変換するため，COBOL プログラムからオブジェクトを識別する手法をまとめている．

この研究によると，COBOL プログラムからオブジェクトを見つけ出す方法は大きく分けて function-driven，data-driven, object-driven の 3 つに分類される．本研究では，バッチ処理システムにおけるデータセットのデータ依存関係に着目しているという点で，data-driven のアプローチに当てはまる．

プログラムスライシング [10] は，プログラム中の特定の文が出力する変数の値に関係を持つプログラム文の集合を計算する技術である．Horwitz ら [11] により，制御依存関係，データ依存関係を用いたグラフ構造上での探索問題として定義されており，本研究では COBOL 用のデータ依存解析として Stap の手法 [12] を用いてグラフの構築を行った．

Ákos ら [13] は，業務システムにおける大規模な COBOL プログラムに対するプログラムスライシング手法を提案している．この手法は制御フローグラフ上においてトークンを伝播させていくというアプローチを用いることで，一般的な制御依存グラフを用いたスライシングと同程度の精度を保ちながら，より効率のよいスライシング計算を実現してる．本研究では，手法において COBOL プログラムのスライシング計算を行うが，対象となるシステムの COBOL プログラムの規模が一般的なスライシング手法であっても問題なく計算できる範囲であったため，Ákos らの手法は用いていない．

Hatano ら [14] は，Java で書かれたシステムにおいて，1 つのメソッド内部に記述されたビジネスロジックの出力に影響を与えるような条件文を特定する手法をプログラムスライシングを使用して実現した．本研究の提案手法は，ジョブ制御言語によって連結された複数の COBOL プログラムの中でのデータフローを抽出しており，また，条件分岐ではなく入出力となるデータ間の関係を抽出する手法となっている．

Meyers ら [15] は，1 つの手続き単位が複数の出力変数を書き出すとき，それぞれの出力変数に対して計算したプログラムスライスが共通の文を多く含むほどその手続きの凝集度が高いと解釈することを提案している．本研究では，COBOL プログラムを出力データごとのプログラムスライスに分解して個別に解析することで，凝集度が低い COBOL プログラムが解析結果の正確さを損なわないようにした．

3. 提案手法

本研究では，バッチ処理を構成するジョブ制御言語の記述と COBOL プログラム群の内容から，そのバッチが行う計算の概要として，バッチ処理全体の初期入力となるデータセットの一覧，最終出力となるデータセットの一覧と，どの入力データセットがどの出力データセットの計算に関係するかを表現した対応表を出力する手法を提案する．

バッチ処理は，ジョブ制御言語で記述され，メインフレームコンピュータ上で実行されるデータ加工の手続きである．システム全体では複数のバッチ処理を実現し，日次や月次といった様々なタイミングで実行するが，本研究では一連のジョブをまとめて実行するジョブネットという単位を 1 つのバッチ処理とみなして解析する．

1 つのジョブネットは，実行するべき関連ジョブを複数（多い場合は数万件）列挙し，それらの実行順序の制約を指定したものである．各ジョブは，さらにその構成要素であるジョブステップの並びによって定義され，業務システムの場合，各ジョブステップの多くは COBOL プログラムから構成される [16]．そのため，本研究ではジョブ制御言語と COBOL プログラムのみを解析対象とする．

本研究では，解析対象となるジョブネット全体をジョブステップの集合 S と，ジョブステップ間に存在する順序制約 $s_i \to s_j (s_i, s_j \in S)$ の集合として取り扱う．順序制約 $s_i \to s_j$ は，s_i の実行が完了してから s_j を実行することを意味しており，たとえば 3 つのジョブステップ s_1, s_2, s_3 に対して $s_1 \to s_3$ かつ $s_2 \to s_3$ であれば，s_1 と s_2 の実行が完了するまでは s_3 の実行は開始されないことを意味する．s_1 と s_2 の間には順序制約が指定されていないので，これらはジョブ管理システムによって並行に実行される可能性がある．ジョブ制御言語の記述では繰り返し処理は存在しないため，順序制約でジョブステップを連結したものは有向非巡回グラフとなる．

バッチ処理を構成するジョブステップ $s \in S$ は，入力データセットの集合 $R(s)$ を使用して COBOL プログラムを実行し，計算結果を出力データセットの集合 $W(s)$ に書き出す処理を行う．各データセットは，複数のデータ項目から構成されるレコードの並びであり，他のバッチ処理からの入力，ジョブステップ間でのデータの受け渡し，他のバッチ処理への出力はすべてデータセットの読み書きとして行われる．使用されるデータセットの名称はジョブ制御言語から取得することができ，読み書きのどちらとしてデータセットを使用するかは COBOL プログラムの記述から得られる．

解析の入力例として，**図 1** に 1 つのジョブネットを構成するジョブ制御言語の記述から抽出されるジョブステップ群とその実行順序を示す．図中で長方形で示されているものがジョブステップであり，それを囲む線がジョブである．辺はジョブステップの実行順序関係を表しており，この図では最初のジョブステップが終了した後，2 つの計算が並行して進行することが示されている．ジョブ制御言語は図 1 左のようにジョブ間の順序を指定するが，本研究ではこれを図 1 右のようにジョブステップを頂点としたグラフとして使用する．図 1 右のグラフでは頂点のラベルとして入出力データセットの名称を付与しており，たとえば最初のジョブステップは，データセット A を入力としてデー

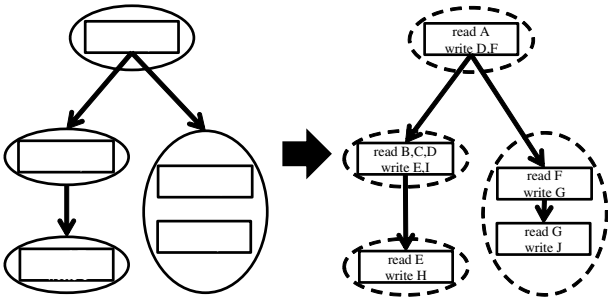

図 1 左はジョブ制御言語で記述されるジョブ間の順序制約の例，右はそれに対応するジョブステップ間の実行順序制約グラフである．長方形がジョブステップであり，ラベルはジョブステップが入出力するデータセットの集合を示す．ジョブステップを囲む破線はジョブ単位を示す．

表 1 提案手法の出力例．初期入力とはジョブネットの内部で出力なしに参照されるデータセット，最終出力はジョブネットの内部で出力された後参照されないデータセットである．

	初期入力 A	初期入力 B	初期入力 C
最終出力 H	✓	✓	✓
最終出力 I	✓	✓	✓
最終出力 J	✓		

タセット D, F を出力することを表現している．

本研究では，バッチ処理に登場するデータセットを，バッチ処理内部での読み書きの情報に基づいて次の 3 つに分類する．

初期入力データセット　あるジョブステップ s において読み込まれるが，順序制約によって s より前に実行されるどのジョブステップの出力データセットにも含まれないようなデータセットを，他のバッチ処理から与えられる入力であるとみなす．

最終出力データセット　あるジョブステップ s において書き込まれた後に，順序制約によって s より後に実行されるどのジョブステップにも読み込まれるようなことがないデータセットを，他のバッチ処理に提供する出力であるとみなす．

中間データセット　あるジョブステップにおいて書き込みが行われた後，他のジョブステップによって読み込みが行われるデータセットは，そのデータセット自体は機能としての意味を持たず，他のデータの生成のための中間的なデータセットとみなす．

上記の分類により，1 つのバッチ処理を，初期入力データセットから最終出力データセットを計算する一連の処理であるとみなす．ただし，ジョブネット内において読み込まれたのち，書き込みが行われるようなデータセットは初期入力データセットかつ最終出力データセットになりうるため，初期入力データセット集合と最終出力データセット集合には共通のデータセットが含まれる可能性がある．一方で，中間データセット集合は，初期入力，最終出力のいずれとも排反である．

図 1 に示したジョブネットが入力として与えられた場合，提案手法は，初期入力データセットが A, B, C，最終出力データセットが H, I, J であると認識し，それらの間の依存関係として**表 1** を抽出する．この表から，これらのデータセットが他のバッチ処理や外部システムと関係する重要なデータセットであり，データセット A, B, C の内容を用いてデータセット H, I の内容を，データセット A の内容からデータセット J の内容を計算していることが分かる．このような情報を複数のバッチから収集していくことで，システム全体で重要な役割を持つデータセットや，それらの関係を明らかにし，システムの再設計の手がかりを得ることができる．提案手法の出力として表形式を採用した理由は，バッチシステムを構成する膨大な数の要素の情報を，抽象度の高い表現に集約するためである．「各ジョブネットがどのデータセットからどのデータセットを生成するか」という情報から個々のバッチの役割や，複数のバッチ間の関係を分析することが，システム全体の構造の理解と再設計に重要であると考えた．バッチシステムにおける各プログラムの振る舞い，たとえば，各プログラムがどのデータセットを扱っているかという情報を出力に付与することや，それらの関係を表現したグラフを出力することは可能である．しかし，これらの情報はシステムの規模（COBOL プログラムの数）に比例して量が増加するため，人間にとって理解が困難となることが予想される．そこで本研究では，ジョブネット単位での動作の概要のみを表形式で出力するものとした．

提案手法は，以下の手順からなる．

(1) ジョブ制御言語の記述から実行順序グラフを作成する．
(2) ジョブステップごとの COBOL プログラムのスライシング情報を使って実行順序グラフを詳細化する．
(3) 実行順序グラフの情報を用いて初期入力データセット，最終出力データセットを識別し，実行順序グラフにそれらを表現するノードを追加する．
(4) 実行順序グラフ上でデータ依存関係を計算し，データ依存グラフを構築する．
(5) データ依存グラフを用いて，初期入力データセットと最終出力データセットの間のデータ依存関係を計算し，表形式で出力する．

なお，ステップ 2 の計算は計算の正確さを向上するために導入しているが必須ではなく，COBOL プログラム中のファイル操作の文を解析するだけでも出力を得ることが可能である．COBOL には様々なコンパイラ依存の言語要素もあり，解析が不可能なファイルなどについてはこの詳細化のステップを飛ばして実行できるようになっている．以降の節では，各ステップの詳細について順番に説明する．

3.1 実行順序グラフの作成

提案手法では，最初にジョブ制御言語の記述内容を解析

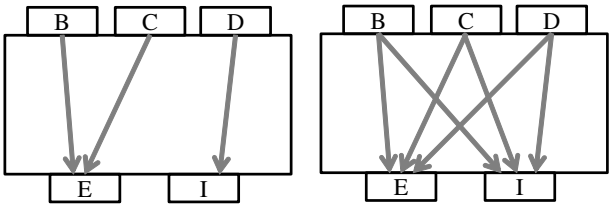

図 2 (左図) 実際のデータ依存関係 (右図) ジョブ制御言語の情報だけから推定されるデータ依存関係

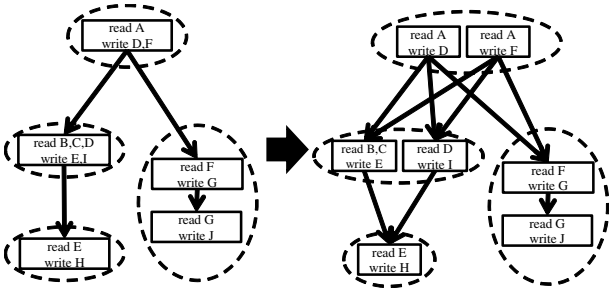

図 3 実行順序グラフの詳細化．左図は詳細化前，右図は詳細化後．

し，実行順序グラフを作成する．実行順序グラフは有向グラフであり，ノードは 1 つのジョブネット内部のすべてのジョブステップ，辺はそれらの間の実行順序の制約に対応する．1 つのジョブ内に定義されているジョブステップは逐次的に実行されるため，ジョブ j_1 の実行後に j_2 を実行するという順序関係は，j_1 の末尾のジョブステップ s_1 と j_2 の先頭のジョブステップ s_2 の間の順序関係として解釈してジョブステップ間の順序関係を接続する．

バッチ処理を構成するジョブステップ $s \in S$ は，ジョブ制御言語の記述で指定されたデータセットの集合を用いて COBOL プログラムを実行する．COBOL プログラム中では，使用するデータセットは OPEN 文によって宣言される．OPEN 文にはオープンモードの指定があり， INPUT は読み込み専用， OUTPUT は書き込み専用，I-O であれば入出力の可能性があることを意味する．これらの情報を正規表現による検索により抽出し，ジョブ制御言語で記述されたすべてのデータセットの集合を，データを読み込む入力データセットの集合 $R(s)$ と計算結果を書き出す出力データセットの集合 $W(s)$ に分類する．これらをグラフのノードのラベルとして付与すると，図 1 に示すようなグラフが得られる．

3.2 COBOL プログラムのスライシング解析による実行順序グラフの詳細化

1 つのジョブステップは複数のデータセットを入力とし，複数のデータセットを出力することができる．一般的には，すべての入力データセットの内容がすべての出力データセットの内容に影響を与えると仮定するのが自然である．しかしながら，業務システムにおける COBOL プログラムは巨大化・複雑化している傾向があり，1 つのプログラムに複数の機能が混在していることがあると経験的に判明している．このような COBOL プログラムでは，直接的に関係のないデータセットの読み込みと書き込みが混在しており，実際には無関係なデータセット間に関係があるとみなしてしまう可能性がある．

図 2 は，無関係なデータセットの間に関係があるとみなされるジョブステップの例を示したものである．このジョブステップはデータセット B，C を読み込んで計算した結果を E に出力し，D を読み込んで計算した結果を I に出力する．2 つの処理は本来は独立であるが，何らかの理由で単一プログラムに処理が記述されてしまうと，本来は図 2 の左図のような入出力関係であるにもかかわらず，ジョブ制御言語の記述と COBOL の OPEN 文からは右図のような入出力関係が推定されてしまう．

本研究ではこの問題を解決するために，COBOL プログラムが複数の出力データセットを持つ場合，**図 3** のように各出力ファイルに対応した部分プログラムへと自動的に分解し，各ノードが高々 1 つの出力データセットを持つグラフへと詳細化を行う．図 2 の例では，B，C を入力として E を出力とする部分プログラムと，D を入力として I を出力とする部分プログラムにノードを分割する．詳細化前のグラフにおいてジョブステップ s_1 からジョブステップ s_2 への辺があった場合は，s_1 に対応するすべてのノード群から s_2 に対応するすべてのノード群へと辺を接続する．図 3 の場合は，A を入力として D，F を出力する 2 つの部分プログラムが先に実行され，そのあとで 3 つのプログラムが実行されるというように 6 本の辺を接続することで，詳細化前の実行順序関係をそのまま保存しておく．

部分プログラムの計算は，プログラムスライシング [10] によって行う．COBOL プログラムでは出力が WRITE 文によって行われているため，1 つのデータセットに対応する WRITE 文の集合をスライシング基準としてプログラムスライスを求めると，WRITE 文で書き込まれるレコードの内容に影響を与える可能性があるすべての文の集合を求めることができる．得られたスライスの中に含まれる READ 文によって読み込まれるデータセットが，その部分プログラムにとっての入力データセットとなる．

COBOL のプログラムスライシングに用いるデータ依存解析には，Stap の手法 [12] を用いた．COBOL における変数は入れ子構造を持つ可能性がある項目として定義され，それ以上小さな項目単位へ分割できない項目は基本項目と呼ばれる．また，基本項目をまとめた最上位の項目をレコード呼ぶ．たとえば，以下の変数の宣言では，DATE がレコードであり，YEAR，MONTH，DAY という項目から構成される．YEAR，MONTH，DAY はそれ以上分割できないため，基本項目である．

```
01 DATE.
    05 YEAR  PIC 9(4).
    05 MONTH PIC 9(2).
    05 DAY   PIC 9(2).
```

この宣言を持つ COBOL プログラムにおいて，レコード DATE への変数の代入が行われるとき，その子となる基本項目すべてが上書きされる可能性がある．そのため，COBOL プログラム中における変数に対するすべての操作を，その変数の子となる基本項目へ展開して解析を行うことで正確さを保っている．また，サブルーチン呼び出しについても対応した解析を行っている．

3.3 初期入力と最終出力の識別

実行順序グラフとして表現されたジョブステップ（あるいはその一部）が行う入出力データセットと，その順序情報が得られているので，それらを使ってジョブネットの初期入力データセット集合 I，最終出力データセット集合 O を求める．I はジョブネット内部で書き込みなしに読み出されるデータセット，O は書き出された結果が読み込まれないデータセットである．具体的には，実行順序グラフ上であるジョブステップ s_1 から s_2 に辺を 1 つ以上たどって到達可能であることを $s_1 \xrightarrow{*} s_2$ と表現すると，I，O は，それぞれ以下のようなデータセット集合に対応する．

$$I = \{i | \exists s.i \in R(s) \wedge \forall s'(s' \xrightarrow{*} s).i \notin W(s')\}$$
$$O = \{o | \exists s.o \in W(s) \wedge \forall s'(s \xrightarrow{*} s').o \notin R(s')\}$$

これらのデータセットはバッチ処理内部に対応する入出力を持たないため，そのままではデータ依存グラフ上に情報が表現されない．そのため，以下のように仮想的な入出力ノードを実行順序グラフに追加する．

- 各初期入力データセット $i \in I$ に対して，すべてのジョブステップよりも先に実行される $entry_i$ ノードを作成し，$R(entry_i) = \phi, W(entry_i) = \{i\}$ とする．
- 各最終出力データセット $o \in O$ に対して，すべてのジョブステップよりも後に実行される $exit_o$ ノードを作成し，$R(exit_o) = \{o\}, W(exit_o) = \phi$ とする．

これにより，すべてのデータセットのデータ依存関係が実行順序グラフ上で計算可能となる．

3.4 データ依存グラフの構築

本ステップでは，ジョブステップ間におけるデータの依存関係を表したデータ依存グラフを構築する．本手順の過程を**図 4**, **図 5** に示す．図 4 は COBOL プログラムの解析の詳細化を行わなかったときの例であり, 図 5 は詳細化を行ったときの例である．それぞれの左図は前ステップまでで求めたラベル付きの実行順序グラフであり，これらから右図のようなジョブステップ間におけるデータ依存グラフを構築する．これらの図において初期入力データセットはデータセット A, B, C であり，最終生成データセットはデータセット H, I, J である．それぞれ $entry$, $exit$ という仮想的な入出力ノードで表現される．

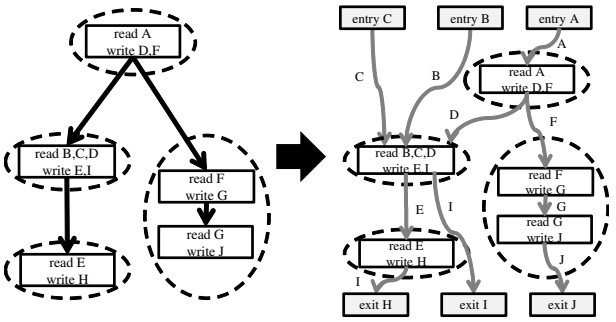

図 4 図 1 のグラフから COBOL プログラムの解析による詳細化なしで計算した場合のデータ依存グラフ．灰色の長方形は初期入力，最終出力データセットに対応する仮想的なノードである．

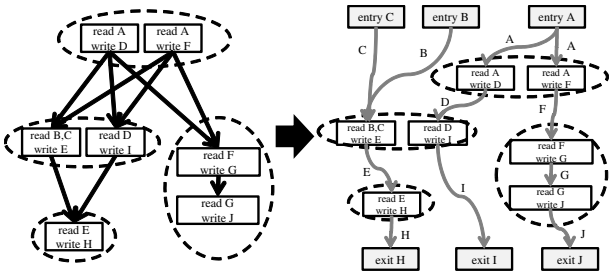

図 5 図 3 のグラフから COBOL プログラムの解析による詳細化ありで計算した場合のデータ依存グラフ．

実行順序グラフにおけるジョブステップ間のデータ依存関係の計算は，一般的なプログラムに対する def-use 関係の計算と以下の 2 点で異なる．1 つ目は，データセットへの出力操作は一般的なプログラムにおけるファイルへの追記に相当する場合があるという点である．通常の def-use の計算であれば変数への値の代入が元の値を上書きすると考えるが，ファイルへの追記は過去に出力された値を上書きしないため，あるジョブステップ s が出力したデータセットの内容は s より後に実行されるすべてのジョブに届くと仮定する．2 つ目は，実行順序グラフ上での経路の分岐は並行での実行を表現しているだけであり，分岐先のすべてのプログラムが必ず実行されるという点である．厳密な def-use を計算するのであれば，一般的なプログラムにおけるマルチスレッド処理の解析と同じように，並行実行されるプログラム間のあらゆる順序関係を考慮する必要がある．この点については，本研究では，同一データセットを扱うジョブステップについて，必ずそれらの実行順序が定義されていることを仮定している．一般には，s_1 と s_2 の間に順序制約がないにもかかわらず $W(s_1) \cap R(s_2) \neq \phi$ が成立するようなジョブ制御言語の記述は可能であるが，ジョブ実行時の順序関係によってデータ依存関係が変化するようなシステムはレガシーシステムとして運用されてい

表 2 データ依存グラフの詳細化を行ったときの提案手法の出力となる対応表の例.

	初期入力 A	初期入力 B	初期入力 C
最終出力 H		✓	✓
最終出力 I	✓		
最終出力 J	✓		

る業務システムには存在しないと仮定している．これら 2 つの仮定に従って，本手法では，以下の条件をすべて満たしたとき，2 つのジョブステップ s_1, s_2 間にデータ依存関係が存在すると判断する．

- s_1 が出力したデータセット d が s_2 の入力データセットである．すなわち，$W(s_1) \cap R(s_2) \neq \phi$ である．
- s_1 が実行された後で s_2 が実行されることがジョブ制御言語で指定されている．すなわち，実行順序グラフ上で $s_1 \xrightarrow{*} s_2$ である．

結果として得られるデータ依存グラフは，実行順序グラフと同じノード集合を持ち，データ依存関係を辺とする有向グラフとなる．実行順序の関係そのものは，データ依存グラフには含めない．

3.5 入出力の対応表の抽出

初期入力データセット $i \in I$ と最終出力データセット $o \in O$ の間に関係がある（o の計算に i が使用されている）かどうかは，データ依存グラフ上で，初期入力データセット i に対応する $entry_i$ ノードから最終出力データセット o に対応する $exit_o$ ノードへのデータ依存辺の推移的な経路が存在するかどうかによって求める．

この対応関係を表 1, **表 2** のような対応表にまとめたものを手法の出力とする．COBOL プログラムの解析の詳細化を行わなかった図 4 のデータ依存グラフから得られる対応表が表 1 であり，詳細化を行った図 5 から得られる対応表が表 2 である．表 1 に示された ✓ が 7 個あるのに対し，表 2 に示された ✓ が 4 個にまで減少している．これは COBOL プログラムをスライシングで分解することにより，最終出力データセットの生成に実際は不必要である初期入力データセットが対応表から取り除かれたことを意味する．

4. ケーススタディ

提案手法を実装し，実際のバッチ処理システムに対し手法を適用して効果を確認するケーススタディを行った．このケーススタディで調査する項目は次の通りである．

(1) ジョブネットにおける初期入力，最終出力，中間データセットの数.
(2) 対応表の大きさに対する，実際に対応関係があるデータセットの組み合わせの割合を，COBOL プログラムの解析による詳細化なしで求めたもの.
(3) (2) と同様の割合を，COBOL プログラムの解析による詳細化ありで求めたとき，(2) の値から減少する度合い.
(4) 対応表の形状の特徴.

表 3 ジョブネットにおける各データセットの数の調査結果．$|I|$ は初期入力データセットの数，$|O|$ は最終出力データセットの数，$|M|$ は中間データセットの数．$|I \cup O \cup M|$ はジョブネットで使用されるすべてのデータセットの総数である．

	$\|I\|$	$\|O\|$	$\|M\|$	$\|I \cup O \cup M\|$
Min.	1.0	1.0	0.0	2.0
Median	81.0	117.0	0.0	308.0
Mean	317.1	283.6	1.5	602.0
Max.	951.0	760.0	117.0	1711.0

(1) はレガシーシステムにおけるジョブネットの規模と特性を調べるものであり，(2) と (3) は対応表の可視化の有効性と，COBOL の解析を導入した効果をそれぞれ定量的に調べるものである．(4) は，得られた対応表の一部に対する目視での定性的な調査である．

ケーススタディは，実際に企業で稼働しているバッチ処理システムのすべてのジョブネットに対し，手法を適用することで実施した．対象システムは金融機関において融資業務を行うものであり，2000 年にメインフレーム上に作成されたものである．ジョブステップに対応するプログラムの多くは COBOL で記述されているが，一部のジョブステップに対応するプログラムはメインフレームコンピュータのユーティリティ・プログラムである．ユーティリティ・プログラムはデータセットのソートやコピーなど，基本的な操作を扱うプログラムであるが，メインフレームコンピュータ依存の処理であるため，今回の評価実験ではそれらのジョブステップは $R(s) = W(s) = \phi$ として扱うことで解析の対象外とした．

以下，各項目の評価結果と考察を順番に述べる．

4.1 ジョブネットにおけるデータセット数

提案手法を用いて，ジョブネットで使用されるデータセットの集合を初期入力データセット集合 I，最終出力データセット O，中間データセット M の 3 つに分類し，それぞれの集合のサイズを計算した．

ジョブネットにおける各データセットの数の最小値，中央値，平均値，最大値を**表 3** に示す．中間データセットの数は他のデータセットの数に比べて小さい傾向が読み取れる．中間データセットはジョブネット内において書き込みが行われた後，読み込みが行われるデータセットである．中間データセットが少ないということは，ジョブネットにおいて，初期入力となるデータセットが読み込まれて何らかの処理が実行され，その結果がそのままジョブネットの最終出力として書き出されるようなデータの流れが多くを占めるということを意味する．

ジョブネット内における一時的なデータセットを中間

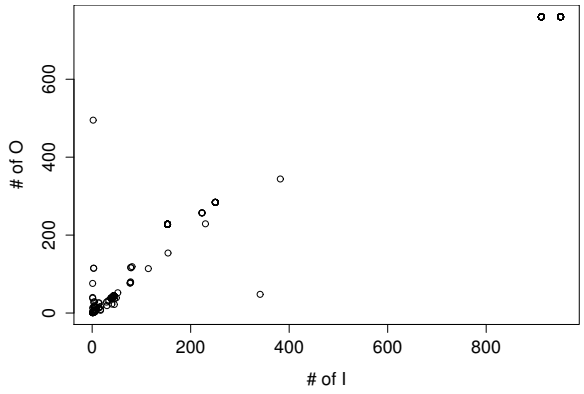

図 6 ジョブネットごとのデータセット数．横軸：初期入力データセット数，縦軸：最終出力データセット数．

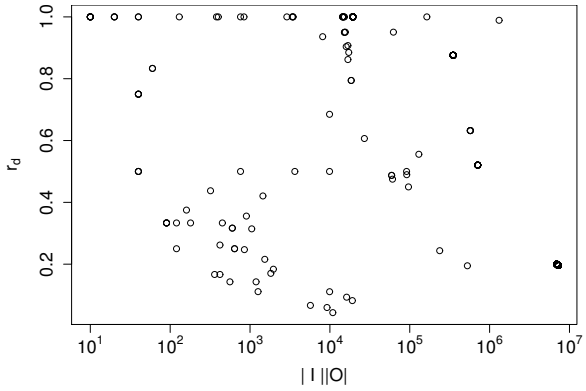

図 7 ジョブネットの規模に対する r_d の値．横軸:対応表の大きさ $|I||O|$，縦軸：r_d の値．

データセットとみなし，対応関係表から除外することで，出力対応表の大きさを抑え込めることを期待していたが，今回の適用対象システムでは，この工夫により実際に出力から除外可能なデータセットは比較的少なく，有効な場合は少ないといえる．

ただし，中には 100 以上の中間データセットをもつジョブネットも存在しており，このようなジョブネットに対しては，すべての書き込みデータセットを出力とする場合にくらべ，100 以上の機能として本質的でない書き込みデータセットを除外することが可能であり，この工夫による貢献が大きいと考えれられる．

また，ジョブネットにおける初期入力データセット数と最終出力データセット数の関係性を散布図で示したものを**図 6** に示す．この図から，ジョブネットにおいて初期入力データセット数と最終出力データセット数の間に正の相関が見られることが確認できる．つまり，ジョブネットの最終出力データセットが多いものは，その生成に必要な初期入力データセットも多い傾向があるといえる．ジョブネットのなかには，初期入力データセット数が最終出力データセット数より極端に大きくなるものや，その逆となるものがある．初期入力データセット数の方が極端に大きいものは，複数のデータセットからひとつのデータセットへデータを集積するような働きをするものであり，最終出力データセット数の方が極端に大きいものは，ひとつのデータセットから複数のデータセットへデータを振り分けているような働きをするものであると考えられる．

4.2 対応表の大きさに対する対応関係があるデータセットの組み合わせの割合

提案手法の出力は，ジョブネットにおける最終出力データセットと，その生成に必要な初期入力データセットの対応表である．最終出力データセットと初期入力データセットのすべての組み合わせに対し，この対応関係が存在するものの割合を調査するため，全ジョブネットに対し，以下の式で定義される割合 r_d を評価した．

表 4 全ジョブネットにおける r_d の値

Min.	Median	Mean	Max.
0.04	0.52	0.59	1.00

$$r_d = \frac{n(I,O)}{|I||O|}$$

ここで，$n(I,O)$ は初期入力データセット集合 I，最終出力データセット集合 O の間で依存関係のある I と O 組み合わせ数である．つまり，$n(I,O)$ は対応表に記されたチェックマーク（✓）の数に一致する．分母の $|I||O|$ は出力となる対応表のマス目の数であり，r_d はチェックマークが埋められている割合に対応する．たとえば，表 1 の例では $r_d = 7/(3 \times 3) = 0.78$ となり，表 2 の例では $r_d = 4/(3 \times 3) = 0.44$ となる．

r_d の値の最小値，中央値，平均値，最大値を**表 4** に示す．対象システムにおいて，r_d の値は平均的に 6 割程度であった．ジョブネットの最終出力となるデータセットに対し，その生成に必要となる初期入力データセットは平均的には全体のうち 6 割程度であるため，あるジョブネットの読み書きするデータセットの集合から，単純に I のすべての要素が O のすべての要素の計算に使われていると解釈してしまうと，中間データセット数が少ないことを考慮に入れたとしても，その半数近くが実際には依存関係のない組み合わせであるということを意味する．したがって，提案手法のようにジョブの実行順序関係を考慮したデータ依存関係を求めることは，実際には依存関係のないデータセットを除外するのに有効であると考えられる．

また，対応表の規模に対する r_d の値を**図 7** に示す．横軸は対応表の大きさ $|I||O|$ であり，対数のスケールになっている．この図からは，対応表の大きさと r_d の相関は見られず，対応表の大きさに対し一様に効果が見られた．つまり，対応表の大きさによらず，手法は有効であると考えられる．今回の対象システムから得られた対応表のなかで，最も大きいものは 72 万個以上のマス目をもつものであった．このジョブネットに対する r_d の値は 0.2 程度であるた

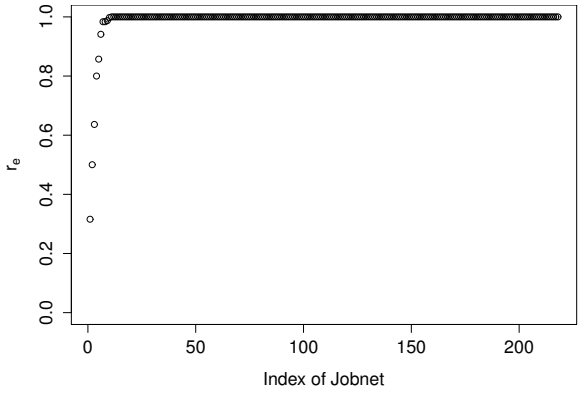

図 8 各ジョブネットの r_e の値．横軸:ジョブネットに対する通し番号，縦軸：r_e の値．

表 5 全ジョブネットにおける r_e の値

Min.	Median	Mean	Max.
0.32	1.00	0.99	1.00

め，単純に I と O のすべての組み合わせを考慮する場合に比べ，60 万近い依存関係の候補を手法により自動的に除外することができた．

4.3 データ依存グラフの詳細化の有効性

本手法には COBOL のスライシングによるデータ依存グラフの詳細化を行うステップが存在する．この詳細化の有効性を調査するため，詳細化を行った場合と行わなかった場合の出力となる対応関係の数の比較を行った．具体的には，全ジョブネットに対し以下の式で定義される r_e の値を求めた．ここで，$n(I,O)$ は詳細化を行わない場合の，$n'(I,O)$ は詳細化を行った場合の初期入力データセットと最終出力データセットの依存関係がある組み合わせ数である．

$$r_e = \frac{n'(I,O)}{n(I,O)}$$

r_e の値は詳細化前後の組み合わせ数の比になっており，この値が 0 に近いほど詳細化の効果があり，1 に近いほど効果がないことを示す．

実験により得られた r_e の値の最小値，中央値，平均値，最大値を**表 5** に，各ジョブネットの r_e の値を**図 8** に示す．r_e の値は中央値で 1.00 となっている．また図 8 から読み取れるように，ほとんどのジョブネットにおいてスライシングによる詳細化の効果が見られなかった．これは対象システムにおける COBOL プログラムにおいて互いに関係のない処理が混在しているものが少ないということを意味しており，対象システムのプログラムの品質としては好ましい状態にあったと考えられる．

一方で，処理が混在している COBOL プログラムを含む一部のジョブネットに対しては，データセット間の関係が半分以下に削減されており，大きな効果があったと考えられる．今回のケーススタディからは，対象システムの COBOL プログラムが明らかに巨大化・複雑化していると分かっている場合にのみ，プログラムスライシングによる詳細化を行うことが有効であるという見解を得た．

表 6 ケーススタディで実際に見られた対応表の例．$i_1 - i_9$，$o_1 - o_7$ がそれぞれ初期入力，最終出力データセット．

	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
o_1	✓	✓	✓	✓					
o_2	✓	✓	✓	✓					
o_3	✓	✓	✓	✓					
o_4					✓	✓	✓		
o_5					✓	✓	✓		
o_6								✓	✓
o_7								✓	✓

4.4 対応表の形状の特徴

4.1 節の結果から，本手法の出力となるデータセットの対応表において膨大な数の初期入力データセットと最終出力データセットが存在する場合があることが分かった．このような場合，人間がその出力から動作の概要を理解することは困難であると考えられる．しかし，複数の対応表の内容を目視で確認したところ，**表 6** に示すように，特定のデータセット間に密なデータ依存関係をもつような対応表が多くのジョブネットから得られていることが判明した．表 6 の例では，最終出力データセット $o_1 - o_3$ が初期入力データセット $i_1 - i_4$ から生成され，同様に $o_4 - o_5$ が $i_5 - i_7$ から，$o_6 - o_7$ が $i_8 - i_9$ から生成される．この場合，ひとつひとつの出力データセットの生成に必要な入力データセットを提示するのではなく，データセットのグループ化を行い，3 つの出力データセット群それぞれに必要な入力データセット群を提示することで，ジョブネットが主に 3 つの処理を扱っているということを明確に表現できる可能性がある．

5. 妥当性への脅威

本研究ではバッチ処理システムの動作の概要を抽出するために，データセットの依存関係に着目するというアプローチとっている．これはデータセットが動作の概要に対応しているという前提によるものであるが，レガシーシステムにおいては，ひとつのデータセットに互いに関係のないデータが混在しているなど，データセット自体が複雑化している場合があり，そのようなシステムに対しては本手法のアプローチが適当でない可能性がある．

本手法ではジョブネット内において書き込みが行われた後，読み込みが行われるデータセットを動作の概要に本質的でない中間データセットと見なすことで，手法の出力から除外している．しかしながら，ジョブ間に定義されている順序関係は実行の前後関係が定義されているにすぎず，実際にバッチ処理として実行されるときには長時間の間隔

が空く場合がある．このような場合，その時点でのデータセットの内容は機能にとって重要な役割を果たすものである可能性がある．本手法ではこのようなデータセットまでも中間データセットとして扱っているため，重要なデータセットを出力から除外してる可能性がある．

提案手法ではジョブ間でやりとりされるデータはデータセットのみであると限定しているが，実際のシステムではデータベースアクセスによるデータの依存関係が生じる可能性があり，本研究のケーススタディではその影響を考慮していない．今回は解析の対象外としたが，アクセスするデータベースのテーブルを仮想的なデータセットとみなし，SQL の SELECT 文で使用されたものを入力データセット，INSERT 文や UPDATE 文で使用されたものを出力データセットとするだけで，手法の解析対象に組み込むことが可能である．

本研究ではジョブが COBOL プログラムで表現されていることを仮定したが，バッチシステムによっては，データセットのコピー操作などのユーティリティプログラムが使用される可能性がある．これらについては，データベースと同様，入出力データセットの表現に変換することができれば提案手法で取り扱うことはできるが，ユーティリティプログラムごとに対応する表現を用意する必要がある．

6. まとめと今後の課題

本研究では，レガシーシステムの 1 つであるバッチ処理システムの動作の概要を理解するため，ジョブネットにおけるジョブの実行順序を考慮し，データセット間の依存関係を抽出する手法を提案した．また，COBOL プログラムのスライシングを用いることでデータ依存関係の詳細化を行う方法も考案した．ケーススタディとして，実際に企業で稼働しているバッチ処理システムに対し手法を適用し，評価実験を行った．結果として，手法の出力から除外する中間データセットは他のデータセット数に比べ比較的少ないこと，データ依存関係においてジョブ間における順序関係を考慮することが有効であることが分かった．また，COBOL プログラムのスライシングによるデータ依存関係の詳細化は，特定のジョブネットにのみ有効であることが分かった．データセットの対応表は規模が大きいが，特定のデータセット集合間に密にデータ依存関係があることが多かったため，データセットのクラスタリングによって対応表を小さく表現できる可能性があることが分かった．ただし，これらの知見は単一のシステムを対象にしたケーススタディの結果であり，より一般的な有効性を評価するため，複数のシステムに対し同様の調査を行う必要がある．

今後の課題として，巨大な対応表に対し，データ依存関係をもとにデータセットのクラスタリングを行うことが挙げられる．類似したデータ依存関係の構造をもつデータセット同士をうまく同じクラスタにまとめることができれば，入力クラスタと出力クラスタ間のデータ依存関係を提示することができ，概要の理解のよりよい支援につながると考えられる．本研究では，手法の出力がデータセット同士の対応表であるため，データセット同士のクラスタリングの方法としては Formal Concept Analysis[17] の適用を検討している．

本研究においてデータベースやユーティリティによる影響は考慮していない．これらの仮定がどの程度出力に影響を与えているのかを定性的に評価することも今後の課題として挙げられる．具体的には，システムに精通した人間による評価や，仕様が明確になっているシステムに対する評価実験などが考えられる．その評価の結果次第では，手法をデータベースやユーティリティ等に対応するように拡張する必要があると考えている．また，提案手法の出力が持つ情報の有用性については，企業内部の有識者からの肯定的な意見を得ているが，実際に分析作業の効率などを測定したわけではない．したがって，提案手法の入出力（外部連携ファイル) が実際に動作の概要を理解することにどれだけ役に立つのかを調査することも，今後の課題として挙げられる．

謝辞　本研究は JSPS 科研費 Nos.26280021, 15H02683 の助成を受けたものです．

参考文献

[1] Bennett, K.: Legacy Systems: Coping with Success, *IEEE Softw.*, Vol. 12, No. 1, pp. 19–23 (1995).

[2] Bisbal, J., Lawless, D., Wu, B. and Grimson, J.: Legacy information systems: issues and directions, *IEEE Software*, Vol. 16, No. 5, pp. 103–111 (1999).

[3] 一般社団法人日本情報システム・ユーザー協会 (JUAS): **ソフトウェアメトリックス調査** *2012*, 一般社団法人日本情報システム・ユーザー協会 (JUAS) (2012).

[4] Sneed, H. M.: Migrating from COBOL to Java, *26th IEEE International Conference on Software Maintenance*, pp. 1–7 (2010).

[5] Mossienko, M.: Automated Cobol to Java recycling, *Proceedings of the Seventh European Conference On Software Maintenance And Reengineering*, pp. 40–50 (2003).

[6] Hasselbring, W., Reussner, R., Jaekel, H., Schlegelmilch, J., Teschke, T. and Krieghoff, S.: The Dublo architecture pattern for smooth migration of business information systems: an experience report, *Proceedings of the 26th International Conference on Software Engineering*, pp. 117–126 (2004).

[7] Peréz-Castillo, R., de Guzmán, I. G. R., Ávila Garcia, O. and Piattini, M.: On the Use of ADM to Contextualize Data on Legacy Source Code for Software Modernization, *16th Working Conference on Reverse Engineering*, pp. 128–132 (2009).

[8] Bao, L., Yin, C., He, W., Ge, J. and Chen, P.: Extracting reusable services from legacy object-oriented systems, *IEEE International Conference on Software Maintenance*, pp. 1–5 (2010).

[9] Wiggerts, T., Bosma, H. and Fielt, E.: Scenarios for the identification of objects in legacy systems, *Proceedings of*

the Fourth Working Conference on Reverse Engineering, pp. 24–32 (1997).

[10] Weiser, M.: Program Slicing, *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449 (1981).

[11] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pp. 26–60 (1990).

[12] Stap, G.: Cobol Data Flow Restructuring, Master's thesis, Free University of Amsterdam, the Netherlands (2005).

[13] Ákos Hajnal and Forgács, I.: A demand-driven approach to slicing legacy COBOL systems, *Journal of Software Maintenance*, Vol. 24, pp. 67–82 (2012).

[14] Hatano, T., Ishio, T., Okada, J., Sakata, Y. and Inoue, K.: Dependency-Based Extraction of Conditional Statements for Understanding Business Rules, *IEICE Transactions on Information and Systems*, Vol. E99-D, No. 4, pp. 1117–1126 (2016).

[15] Meyers, T. M. and Binkley, D.: An Empirical Study of Slice-based Cohesion and Coupling Metrics, *ACM Transactions on Software Engineering Methodology*, Vol. 17, No. 1, pp. 2:1–2:27 (2007).

[16] 神居俊哉: メインフレーム実践ハンドブック, リックテレコム (2009).

[17] Ganter, B. and Wille, R.: *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag New York, Inc. (1997).