ソースコード解析に基づくレガシーソフトウェアへのSOA適用手法

Applying SOA to Legacy Software Based on Source Codes Analysis

木 村 隆 洋

要 約 サービス指向アーキテクチャ（SOA）は，分散したシステムを柔軟に連携・統合するための，新しいシステムアーキテクチャとして注目されている．しかしながら，レガシーシステムに対してSOAを適用することは容易ではない．レガシーシステムへの効率的なSOA適用を支援するため，本稿ではレガシーソフトウェアからサービスの候補を抽出する手法を提案する．提案手法では，レガシーソフトウェアのソースコードを解析し，依存の強い処理群をサービスとして抽出する．また，提案手法の有効性を確認するために，既存のアプリケーションからサービスを抽出するケーススタディを行った．その結果，既存アプリケーションのソースコードからサービスの候補を抽出可能であることを確認した．

Abstract The service-oriented architecture (SOA) is a new system architecture that allows flexible integration and orchestration of distributed heterogeneous systems. However, it is not straight forward to apply SOA to existing legacy systems. To support sufficient adaptation of legacy systems for SOA, this paper presents a method that systematically extracts services candidates from source codes of a procedural system. With the proposed method, we analyze source codes of legacy applications and extract highly dependent processes as a service. To illustrate the effectiveness of the proposed method, we have conducted a case study for an existing legacy application. As a result, it was shown that services candidates were identified reasonably from the legacy application.

1. はじめに

今日，ビジネス環境はますます急激に変化するようになり，企業の業務システムもその変化に対して，迅速かつ柔軟に対応することが求められている[8]．しかし，長年に渡り保守されてきた現役の「レガシーシステム」は，業務毎に高度に専門化されたモノリシック（一枚岩）なシステムが多く，任意のシステムとの相互運用性やデータの共有などが考慮されていない．このようなシステムでは，業務手順（業務プロセス）が変わる度にシステムの大幅な修正が必要となり，保守コストが膨大になることが大きな問題とされている．

このような背景の下，サービス指向アーキテクチャ（SOA）[7]というアーキテクチャパラダイムが注目されている．SOAでは，システムの機能を「サービス」という単位でくくりだし，業務プロセスの基本構成要素（要素プロセスと呼ぶ）と対応させる．これらの「サービス」は，システムのプラットフォームや内部実装に非依存なインタフェースを通じて疎結合[14]される．ビジネスプロセスは既存サービスの緩い結合により構成され，個々のサービスの組み換えや更新は容易である．その結果，ビジネス環境の変化に迅速・柔軟に対応することができる．最近では特に，このような利点から，レガシーシステムにSOAを適用する話題に大きな関心が集まっている．

SOAに基づくシステム開発では，まずビジネス環境を分析して要素プロセスを決定した後，これらの要素プロセスに合わせて，システム機能をサービスとして対応付けて開発する方法が

知られている[6]．しかしながら，レガシーシステムには，長年の保守による保守性の低下[4][5]や，変更による業務への影響が許されない，というような制約がある．このため，ビジネス環境に合わせた大幅な修正を行うことは困難になっている．従って，レガシーシステムにSOAを適用する際には，既存システムの機能を最大限再利用することを考慮し，ビジネスの要素プロセスを決定する方法が有効であると考えられる．具体的には，まず，既存のシステムを分析し，サービスの候補を抽出する．次に，ビジネス環境を分析して要素プロセスを決定する．そして，システムから抽出したサービスの候補と，ビジネス環境から抽出した要素プロセスをすり合わせることで，現実的な形でシステムにSOAを適用することが可能となる．この手法では，システムの機能を極力再利用する形でサービスを抽出することが求められる．

そこで本稿では，既存のシステムのソースコードを分析し，サービス（候補）を抽出する手法を提案する．具体的には，まずソースコードに対してリバースエンジニアリングを適用し，データフローダイアグラム（DFD）を取得する．次に，DFD上のデータを三つのカテゴリに分類し，プロセス間の依存関係を4種類に分類，設定する．この依存関係に基づき，DFD上の複数のプロセスをサービスとして抽出する四つのルールを提案する．

また，ケーススタディとして，提案手法を酒在庫管理システム[12]の複数の実装に適用し，サービス抽出を行った．その結果，酒在庫管理システムのソースコードからサービスを抽出することができた．

2. 準備

2.1 サービス指向アーキテクチャ（SOA）

サービス指向アーキテクチャ（SOA）[7]とは，ソフトウェアの機能を「サービス」という単位でくくり，複数のサービスを連携・統合することでシステムを構築するソフトウェアアーキテクチャである．ここで，サービスとはソフトウェアで実行される処理の集合を指し，サービス提供者によってサービス利用者に対して提供される．サービス利用者は，実現したい業務プロセスに合わせて，公開されたサービスを自由に組み合わせて利用することができる．また，一度公開したサービスのインタフェースは原則として変更することは認められないため，サービスを提供するシステム内部に変更があった場合でも，その変更がサービス利用者に影響を及ぼすことはない．図1にSOAに基づくシステムのイメージを示す．Xシステムはサービス A，サービス Bを，Yシステムはサービス C，サービス Dを公開しているとする．この場合に，サービス利用者は公開されたサービスを自由に利用することができる．システム Xのサービスを利用する業務プロセス α，システム Yのサービスを利用する業務プロセス β とすることも可能であるし，両方のサービスを統合的に利用することも可能である．業務プロセスの変更はサービスの組み替えにより実現される．よって，業務プロセス α，業務プロセス β を統合して業務プロセス $\alpha\beta$ とした場合であっても，サービスそのものやシステムに業務プロセスの変更が波及することはない．また，システム Yをシステム Y’に変更した場合であっても，その変更はサービス C，サービス Dのインタフェースには及ばないため，サービス利用者にその変更が波及することはない．このようにして，SOAに基づくシステムでは複数のソフトウェアの機能を疎結合[14]することにより，ビジネス環境の変化に伴うシステムの変更に柔軟に対応することが容易となる．

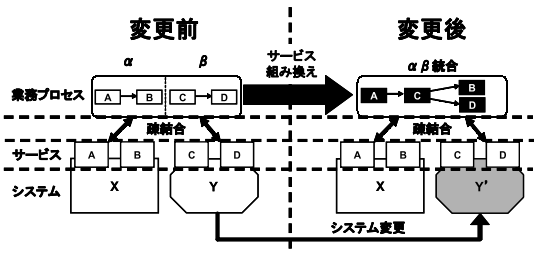

図 1　SOA に基づくシステム

2.2　サービスの条件

サービスの定義に関しては様々なものが存在する[2][3]が，本稿ではサービスを自己完結，オープンなインタフェース，任意の粒度の三つの条件を満たすソフトウェア処理（タスク，プロセス）の集合と位置づける．これらの条件は，SOA における一般的なサービスに要求される必要条件である．以下にサービスのそれぞれの条件について述べる．

（条件 1：自己完結）

サービスは他のサービスに依存せずに実行可能である．つまり，サービスは単独で実行することや，繰り返し実行すること，他のサービスと自由に組み合わせて利用することができなければならない．よって，他の処理を実行した後でなければ実行できない処理や，入力として他の処理の出力が必須となる処理はサービスとして認められない．

（条件 2：オープンなインタフェース）

サービスは，外部から利用可能なオープンなインタフェースを備える．オープンなインタフェースとは，サービスを公開するシステムの実装に依存せず，サービス利用者のデータまたは他システムのデータを入出力データとして指定可能なインタフェースのことである．このため，プラットフォームや言語に依存する呼び出し方法を要求するような処理や，サービスを提供するシステム内部でしか利用されない一時データを入出力とするような処理はオープンなインタフェースを備えていない．

（条件 3：任意の粒度）

サービスの粒度は任意である．条件 1，条件 2 を充足するサービスは粒度が粗くなる（粗粒度）傾向があるが，サービスを利用する業務によって最適な粒度は異なる．このため，業務処理に応じて任意のサービスの粒度を選択できるようにすることが必要である．

2.3　レガシーシステムへの SOA の適用

牧野が提案している SOA に基づくシステム開発手法[3]の概要を図 2 に示す．この手法では，まずビジネス環境のフロー分析を行い，業務の流れを抽象化・単純化した業務プロセスとして表現する．さらに，業務プロセスをそれ以上細分化できない「要素プロセス」まで詳細化する．次に，システム化の側面から分析を行い，要素プロセスに対応付ける形でシステムを構成する細かいコンポーネントやプログラムを適切な粒度のモジュールにまとめ，サービスとする．図 2 では業務プロセスから要素プロセス A，要素プロセス B，要素プロセス C，要素プロセス D

を抽出しており，それぞれにサービス A，サービス B，サービス C，サービス D を対応付けている．そして，サービス A，サービス B を提供する新規システム X，サービス C，サービス D を提供する新規システム Y を開発する．この手法は業務プロセスに合わせてシステムを開発するという意味で，トップダウンなアプローチと言うことができる．

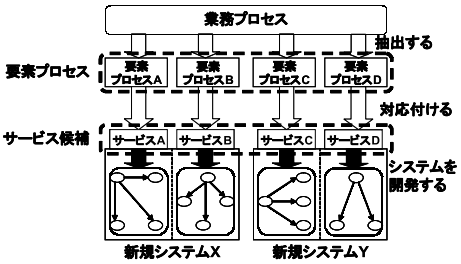

図 2　SOA に基づくシステム開発

このような業務プロセスを中心としたトップダウンなアプローチは，新規に SOA に基づくシステムを開発することを想定している．このため，既にシステムが存在しているレガシーシステムに SOA を適用する場合には，このアプローチを直接適用することは難しいと考えられる．本稿では，レガシーシステムをレガシーソフトウェアにより構成されているシステムと定義する．レガシーソフトウェアとは，長年に渡って運用，保守され続けているソフトウェアのことである．先に述べた SOA に基づくシステム開発手法では，ビジネス環境のプロセスモデル化はレガシーシステムの実装を考慮して行われるとは限らない．このため，業務プロセスに対応するサービスをレガシーシステムから抽出するためには，システム側の大幅な修正が必要となる可能性がある．しかし，レガシーシステムは，長年に渡る保守作業の結果，更なる修正が難しい状態になっており[4][5]，また，現在も業務で使用されているため大幅な改修を加えることが難しい．

従って，レガシーシステムに SOA を適用する場合には，システムの実装を中心としたボトムアップなアプローチの方が現実的であると考えられる．図 3 にその概要を示す．このアプローチでは，まずシステムの分析を行い，現行のシステム機能をできるだけ再利用する形でサービスの候補を抽出する．次に，業務フロー中心のアプローチと同様に業務プロセスを詳細化した要素プロセスを抽出する．そして，サービスの候補と要素プロセスのすり合わせを行う．図 3 では既存システム X ’からサービス A'とサービス B ’を，既存システム Y'からサービス C'とサービス D ’を抽出する．その上で業務フロー中心のアプローチと同様に，業務プロセスから要素プロセス A，要素プロセス B，要素プロセス C，要素プロセス D を抽出する．このようにして抽出したサービスと要素プロセスをすり合わせ, 現実的な対応方法を模索するのである．

このようにして，事前にレガシーソフトウェアから抽出可能なサービスを明らかにすることで，業務フローの分析を行う際の参考や方針決定の材料にすることが可能となる．例えば，業務フローから抽出可能な要素プロセスの候補が複数ある場合には，レガシーソフトウェアから抽出したサービスが使いやすくなるようなものを選択することが可能となる．また，レガシーシステムの実装を極力変更したくない場合には，あらかじめ抽出したサービスの候補を指針として，業務フローから要素プロセスを抽出することが可能であろう．逆に，業務フローを基準とする必要があり，そのためにレガシーソフトウェアを改修しなければならない場合にも，抽

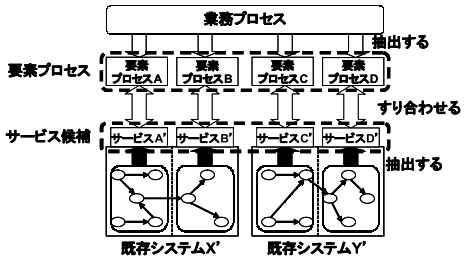

図 3　レガシーシステムへの SOA 適用手順

出したサービスと要素プロセスの差分が明確になるので，改修方針の決定やコスト，期間の見積もりに際して利用することが可能であろう．

2.4　レガシーシステムからのサービス抽出

レガシーシステムを中心としたボトムアップなアプローチを採用する場合には，いかにして既存のシステムからサービスの候補を抽出するかが課題となる．レガシーソフトウェアは変更が難しいという制約があるため，極力レガシーソフトウェアを再利用する形でサービスを抽出する必要がある．しかし，現状ではレガシーシステムからサービスを抽出するための体系的な手法は存在しない．そこで，レガシーシステムへの SOA 適用を支援するために，本稿ではレガシーソフトウェアのソースコードを入力として，2.2 節の条件 1〜3 を満たすサービスの候補を出力する手法を提案する．

提案手法の意義は，次のとおりである．まず，レガシーソフトウェアのソースコードを入力としているため，保守が滞りがち[13]な要求仕様書や設計書等の関連ドキュメントは不要である．また，現行のソースコードからサービスの候補を抽出することができるため，システムの実装を正確に反映したサービスの候補を得ることができる．こうして得られたサービスの候補を基に，現行のシステムを最大限考慮した実現可能性の高い要素プロセスの決定・対応付けが可能である．なお，提案手法は手続き型言語を対象としているが，COBOL や C 言語のような手続き型言語により記述されたレガシーソフトウェアは数多く存在するため，レガシーシステムの現状に即していると考えられる．

3.　提案サービス抽出法

3.1　キーアイデア

提案手法のキーアイデアは，データフローダイアグラム（DFD）を用いて，ソースコード上の処理間に存在する依存関係を解析し，サービスとして成立する処理群を抽出することである．DFD は処理間のデータフローを表現するものであり，データの流れに着目して処理間の依存関係を解析するのに適している．また，DFD は階層毎に適用することが可能であるため，様々なレイヤで処理間の依存解析を行うことが可能である．提案手法は以下の四つのステップで構成される．

（STEP 1）ソースコードを DFD に変換する．
（STEP 2）DFD 上の処理間を流れるデータを分類する．
（STEP 3）DFD 上の処理間に依存関係を設定する．
（STEP 4）DFD に処理結合ルールを適用する．

提案する STEP 1 から STEP 4 までの流れを図 4 に示す．なお，（STEP 1）については，手続き型のソースコードをリバースエンジニアリングし，階層的 DFD を導出する手法[1][6]が提案されている．よって，詳細は本稿では割愛する．導出された DFD の各レイヤに対して，（STEP 2）以降を適用する．DFD の表記法として，処理（以降プロセスと呼ぶ）を円で，プロセス間のデータの流れ（データフロー）を実線矢印，データストアを平行線で表す．

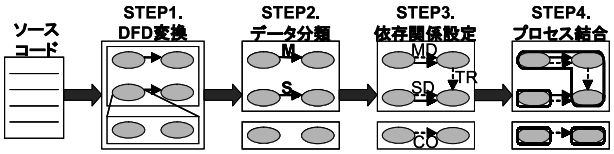

図 4　提案手法の流れ

3.2　データの分類（STEP 2）

サービスのオープンなインタフェース（2.2 節の条件 2 を参照）を実現するためには，プロセス間を流れるデータの種類を明らかにし，サービスの入出力にすることができないデータをサービス内に隠蔽する必要がある．ここでは，プロセス間のデータを外部データ，モジュールデータ，システムデータの 3 種類に分類する．

1）　外部データ（External Data）

外部データとは，システムの外部要素（アクター）とシステム内部のプロセスとの間でやり取りされるデータのことである．外部データが成立するための要件は，アクターとプロセス間でデータフローが発生することである．プログラムコード上では，ファイルや標準入出力，外部コンポーネントとの通信により取得したデータなどが該当する．DFD 上では，データフローに E とラベル付けする．

2）　モジュールデータ（Module Data）

モジュールデータとは，システム内部の特定のプロセスが一時的に使用するデータのことである．モジュールデータは同一のレイヤに属するプロセス間でのみ利用することが可能であり，他のレイヤに属するプロセスからはそのデータは使用不可能である．モジュールデータが成立するための要件は，ⅰ）プロセス間のデータフローが存在する，ⅱ）プロセスとデータストア間のデータフローが存在し，そのデータストアは特定のレイヤ内でのみ有効である，のいずれかを満たすことである．プログラムコード上ではローカル変数や，グローバル変数のうちスコープが対象システムのソースコードの一部にのみ及ぶものなどが該当する．DFD 上では，データフローに M とラベル付けする．

3）　システムデータ（System Data）

システムデータとは，システム内部の全てのプロセスが共通して利用するデータのことである．システムデータは異なるレイヤに属するプロセス間で利用することが可能である．システムデータが成立するための要件は，プロセスとデータストア間のデータフローが存在し，データストア上のデータは全レイヤのプロセスから利用が可能であることである．プログラムコー

ド上ではデータベースに入出力するデータや，グローバル変数のうちスコープが対象システムのソースコード全体に及ぶものなどが該当する．DFD 上では，データフローに S とラベル付けする．

3.3　依存関係の設定（STEP 3）

サービスの自己完結性（2.2 節の条件 1 を参照）を実現するためには，DFD のプロセス間の依存関係を解析し，依存関係の強い処理群をサービス内に隠蔽する必要がある．依存関係を解析する際に，STEP 2 のデータフロー分類を利用する．

DFD はプロセス間のデータフローを解析するためのものであり，プロセス間の依存関係を表現することはできない．DFD を拡張したものとして，Ward の変換図[9]が提案されている．Ward の変換図では，プロセス間の依存関係を破線矢印により表現できる．しかし，Ward の変換図はもともとプロセスの依存関係の分類を行うことを目的としていないため，依存関係の種類を明示的に指定できない．そこで，Ward の変換図を拡張し，プロセス間の依存関係を発生原因毎に分類することで，サービス抽出に利用する．

提案手法では，プロセス間の依存関係をモジュールデータ依存，処理依存，システムデータ依存，制御依存の 4 種類に分類する．以下にそれぞれの依存関係について述べる．以降，P 1, P 2 を任意のプロセスとする．

1)　モジュールデータ依存（Module Data Dependency）

モジュールデータ依存とは，モジュールデータの入出力により発生する依存関係のことである．P 1 の出力データが P 2 の入力データになっている場合，P 1 から P 2 へのデータ依存を設定する．モジュールデータのやり取りによるデータ依存をモジュールデータ依存と呼ぶ．DFD 上では P 1 から P 2 に破線矢印を引き，MD とラベル付けする．図 5 は DFD にモジュールデータ依存をラベル付けしたものである．モジュールデータ依存が成立するための要件は，図 5 左の例のように，P 1, P 2 間で直接モジュールデータの入出力によるデータフローが発生していること，または図 5 右の例のように，データストアを介して，モジュールデータの入出力によるデータフローが発生していることである．

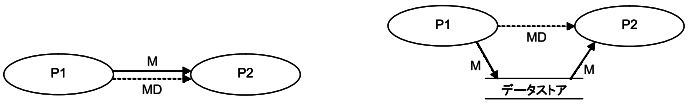

図 5　モジュールデータ依存の例

2)　処理依存（Transaction Dependency）

処理依存とは，プロセス間の処理内容により発生する依存関係のことである．P 1 と P 2 の両方を同時に実行しなければならない場合，P 1 と P 2 の間に処理依存を設定する．DFD 上は P 1 と P 2 の間に破線矢印を引き，TR とラベル付けする．図 6 は DFD に処理依存をラベル付けしたものである．処理依存が成立するための要件は，P 1 と P 2 を同一のトランザクション内で実行する必要があることである．ここでいうトランザクションとは，一貫性を保持した形で複数の処理を実行することである[11]．図 6 左の例のように，アクター B にデータ出力する

ための前提としてアクター A へのデータ出力が必要な場合や，図 6 右の例のように，データストア A とデータストア B のどちらを先に更新しても良いが，同一のトランザクション内で実行することが必要な場合などが処理依存に該当する．前者の場合には破線矢印は P 1 から P 2 への一方向とし，後者の場合には P 1, P 2 間の破線矢印は双方向とする．

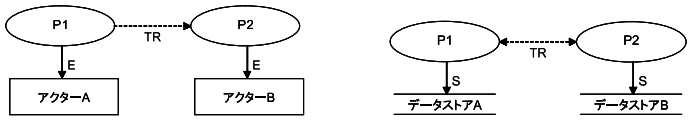

図 6　処理依存の例

3）　システムデータ依存（System Data Dependency）

システムデータ依存とは，システムデータの入出力により発生する依存関係のことである．P 1 の出力データが P 2 の入力データになっている場合，P 1 から P 2 へのデータ依存を設定する．システムデータのやり取りによるデータ依存をシステムデータ依存と呼ぶ．DFD 上は P 1 から P 2 に破線矢印を引き，SD とラベル付けする．図 7 は DFD にシステムデータ依存をラベル付けしたものである．システムデータ依存が成立するための要件は，P 1, P 2 間でデータストアを介して，システムデータによるデータフローが発生していることである．

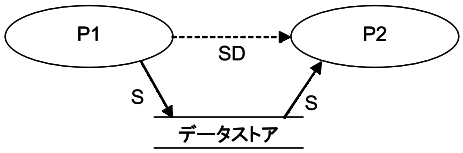

図 7　システムデータ依存の例

4）　制御依存（Control Dependency）

制御依存とは，プロセス間の制御により発生する依存関係のことである．P 1 の出力によって，P 2 の実行の有無を決定する（つまり，P 1 が P 2 の制御フラグとなる）場合，P 1 と P 2 の間に制御依存を設定する．DFD 上は P 1 から P 2 へ破線矢印を引き，CO とラベル付けする．図 8 は DFD に制御依存をラベル付けしたものである．制御依存が成立するための要件は，P 1 の実行結果により P 2 の実行有無に影響が及ぶ実装になっていることである．

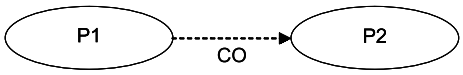

図 8　制御依存の例

プロセス間に複数の依存関係が認められる場合には, より強い依存関係を優先して設定する．依存関係の優先順位を図 9 に示す．最も強い依存関係はモジュールデータ依存である．次に強い依存関係は処理依存であり，その次がシステムデータ依存，最も依存関係が弱いのは制御依存である．例えば，P 1 と P 2 の間にシステムデータ依存が設定するためには，P 1 と P 2 の間にモジュールデータ依存，処理依存が存在しないことが必要となる．

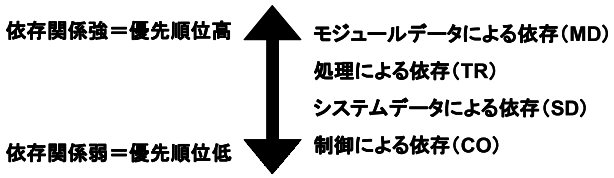

図 9　依存関係の優先順位

3.4　サービスの抽出（STEP 4）

STEP 3 で設定した依存関係を利用し，依存関係の強いプロセス群を結合し，サービスの候補として抽出する．DFD 上のプロセス群をサービスとして抽出するためには，対象となるプロセス群がサービスの条件を満たしていなければならない．2.2 節で述べたとおり，サービスの条件には，自己完結（条件 1），オープンなインタフェース（条件 2），任意の粒度（条件 3）の三つがある．提案するサービス抽出ルールは，サービスの条件 1 とサービスの条件 2 を充足する形で DFD 上のプロセス群を結合するものである．提案手法を任意の DFD レイヤに適用し，任意の粒度のサービスを抽出可能とすることにより，サービスの条件 3 を充足することができる．

P 1 と P 2 を一つのサービス内に結合する必要がある場合，両者の結合を結合プロセスと呼び，P 1＋P 2 と書く．P 1 と P 2 が分割可能な場合，これらを分割プロセスと呼び，P 1｜P 2 と書く．なお，サービスは自由に組み合わせて利用することが可能であるため，複数の分割プロセスを結合して一つのサービスとすることも可能である．以下に，プロセスの結合要否を判断する四つのルールを挙げる．

（ルール 1）モジュールデータ依存プロセスの結合

モジュールデータによる依存が存在するプロセスは，結合が必要である．図 10 に本ルールの適用結果を示す．サービスの条件 1 を実現するためには，サービスを自由に組み合わせて実行することが可能でなければならない．しかし，モジュールデータはシステム内の同一レイヤに属するプロセス間でのみ利用可能な，実装に極めて依存したデータである．P 1 と P 2 の間にモジュールデータ依存が存在する場合に，P 1 と P 2 を別々のサービスとして分割してしまうと，サービス利用者が P 1 サービス，P 2 サービス間の入出力データの受け渡しを代行しなければならない．これでは P 1 サービス実行後に P 2 サービスを実行しなければならなくなり，サービスの条件 1 に反する．P 1 と P 2 を結合することにより，決まった順序で実行が必要なプロセスをサービス内部に隠蔽し，サービスを自由に組み合わせて利用することが可能となる．また，サービスの条件 2 を実現するためには，システム内部のデータをサービスの入出力から排除する必要がある．P 1 と P 2 を別々のサービスとして分割してしまうと，モジュールデータをサービスの入出力データとしなければならなくなる．モジュールデータはシステム内部のデータであるため，サービスの入出力とすることはサービスの条件 2 に反する．P 1 と P 2 を結合することにより，モジュールデータをサービスの入出力から排除することが可能となる．以上より，P 1 と P 2 は結合する必要があり，P 1＋P 2 が成立する．

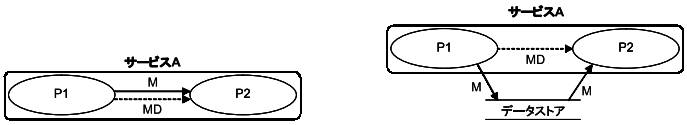

図 10　モジュールデータ依存プロセスの分割

（ルール 2）システムデータ依存プロセスの分割

システムデータによる依存が存在するプロセスは，分割してそれぞれ別のサービスとすることが可能である．図 11 に本ルールの適用結果を示す．P 1 と P 2 の間にシステムデータ依存が存在する場合であっても，P 1 はデータを任意のタイミングでデータストアに出力することが可能である．システムデータはシステム内の全てのプロセスから参照が可能であるため，P 1 が適切なデータを出力した後であれば，P 2 はそのデータを任意のタイミングで取得することが可能である．よって，P 1 と P 2 は互いに依存することなく非同期に実行することが可能である．このため，P 1 と P 2 を別々のサービスとして分割しても，P 1 と P 2 を非同期に実行することが可能であり，サービスの条件 1 を充足する．また，システムデータはシステム内部のデータであるため，サービスの入出力にするとサービスの条件 2 を実現することができなくなる．しかし，システムデータは異なるレイヤに属するプロセス間で利用が可能である．そのため，P 1 と P 2 を別々のサービスとして分割しても，P 2 は P 1 が出力したデータを取得することが可能である．よって，P 1 と P 2 の入出力データとしてシステムデータを指定することなくデータの受け渡しが可能となり，サービスの条件 2 を充足する．以上より，P 1 と P 2 は分割が可能であり，P 1 | P 2 が成立する．なお，必要に応じて，P 1 と P 2 を結合して，1 つのサービスとすることも可能である．

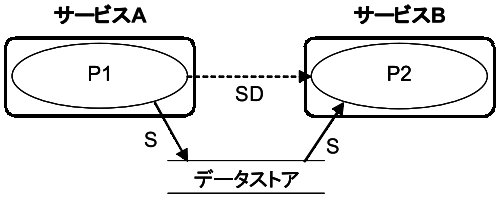

図 11　システムデータ依存プロセスの分割

（ルール 3）処理依存プロセスの結合

処理による依存が存在するプロセスは，結合が必要である．図 12 に本ルールの適用結果を示す．P 1 と P 2 の間に処理依存が存在する場合，P 1 と P 2 は同一のトランザクション内で実行しなければならない．P 1 と P 2 を別々のサービスとして分割した場合，利用者が P 1 と P 2 を同一のトランザクション内で実行することが必要となる．このことはサービスを自由に組み合わせて利用することが不可能であることを意味し，サービスの条件 1 に反する．P 1 と P 2 を結合することにより，同一トランザクション内で実行が必要な処理をサービス内に隠蔽し，サービスの自由な組み合わせを実現することが可能になる．また，処理依存が存在するプロセス間のデータフローには，システムデータの入出力によるデータフローのみが存在する．ルール 2 で述べたとおり，システムデータは入出力データとして指定する必要はない．このため，P 1 と P 2 を別々のサービスとして分割しても，サービスの条件 2 を充足することが可能であ

る．以上より，P 1 と P 2 は結合する必要があり，P 1＋P 2 が成立する．

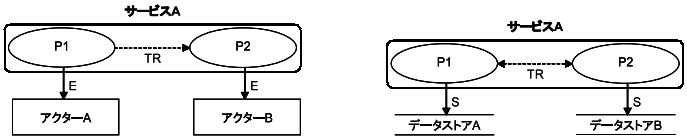

図 12　処理依存プロセスの結合

(ルール 4) 制御依存プロセスの分割

制御による依存が存在するプロセスは，分割してそれぞれ別のサービスとすることが可能である．図 13 に本ルールの適用結果を示す．P 1 の出力は P 2 を実行するか否かを判断するための制御フラグであり，P 2 の入力データにはならない．つまり，P 1 は P 2 を実行するための必要条件を満たすわけではなく，実行の有無を判断しているだけである．このため，P 2 を実行するための条件が整っているのであれば，P 1 を実行せずに P 2 を実行することも可能である．よって，P 1 と P 2 を分割してサービスとした場合であっても，P 1 と P 2 を自由に組み合わせて実行することが可能であり，サービスの条件 1 を充足する．また，制御依存が存在するプロセス間にはデータフローは存在しない．このため，P 1 と P 2 を別々のサービスとして分割した場合であっても，サービスの条件 2 を充足する．以上より，P 1 と P 2 は分割が可能であり，P 1 | P 2 が成立する．

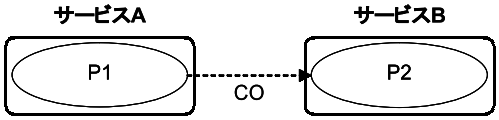

図 13　制御依存プロセスの分割

4.　ケーススタディ

4.1　酒在庫管理システム

提案手法の有効性を確認するために，酒屋の在庫問題[12]の一実装に提案手法を適用し，サービスの候補の抽出を行った．酒屋の在庫問題はプログラミング教育などで利用すべく，共通の例題として提案されたものである．酒屋の在庫問題では，酒屋の倉庫管理が主題となっている．本稿では，酒屋の在庫を管理するシステムを酒在庫管理システムと呼ぶ．

酒在庫管理システムの主な機能として，入庫管理，出庫管理，在庫待ち解消，在庫不足管理の 4 つがある．本稿ではそのうち入庫管理機能と出庫管理機能に焦点を当てて説明する．入庫管理機能の概要を図 14 に，出庫管理機能の概要を図 15 に示す．入庫管理とは，(1)酒倉庫に搬入した酒の在庫を(2)蓄積する機能である．出庫管理とは，客から注文を受けて(1)搬出した酒の在庫を(2)更新し，(3)出庫した酒の情報を通知する機能である．

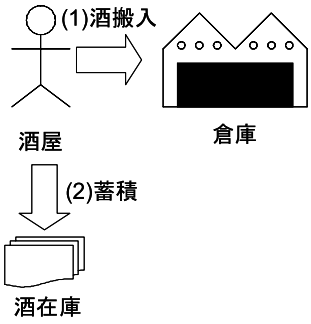

図 14　出庫管理機能の概要

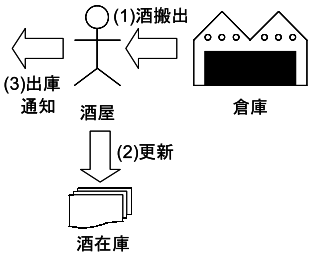

図 15　出庫管理機能の概要

4.2　酒在庫管理プログラム全体への適用

まず，酒在庫管理プログラムの実装から取得した DFD の上位レイヤに対して提案手法を適用し，酒在庫管理システム全体からサービス抽出を行った．図 16 は酒在庫管理プログラムに提案手法を適用した結果である．酒在庫管理プログラムは，(1)入庫依頼を入力として入庫管理処理を行う，(2)出庫依頼を入力として出庫管理処理を行う，という 2 つのプロセスから構成されている．(1)と(2)の間にはシステムデータ依存が存在するため，(ルール 2)により分割が可能である．他に適用可能なルールはないので，(1)｜(2)が酒在庫管理システムの上位レイヤから抽出可能なサービスである（図中ではサービスを角丸枠で囲む）．それぞれ，「入庫管理サービス」，「出庫管理サービス」，となり，本実装ではシステムの主な機能が，2 つの依存関係の弱いプロセスとして分割されていたことが分かる．さらに粒度の細かいサービスを抽出したい場合には，下位レイヤに対して提案手法を適用する．

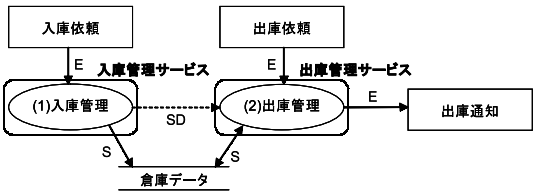

図 16　酒在庫管理プログラム（プログラム全体）から抽出可能なサービス

4.3　入庫管理プロセスへの適用

図 16 の(1)入庫プロセスの下位レイヤからサービスを抽出した．図 17 に提案手法の適用結

果を示す．入庫管理プロセスは，(1)入庫依頼から必要なパラメータを切り出す，(2)切り出したパラメータの正当性チェックを行う，(3)チェックの結果問題がなければ酒在庫データにデータを蓄積する，という三つのプロセスから構成されている．(1)と (2)，(1)と (3)の間にはモジュールデータ依存が存在するため，(ルール 1)を適用して結合する．また，(2)と (3)の間には制御依存が存在するため，(ルール 4)を適用して分割することが可能である．以上より，(1)＋(2)｜(1)＋(3)の 2 つの結合プロセスがサービスとして抽出された．すなわち，入力チェックサービスと酒在庫更新サービスは依存が弱いため，分割することが可能である．

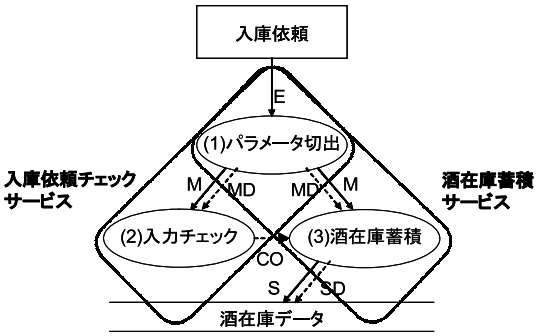

図 17　入庫管理プロセスから抽出可能なサービス

4.4　出庫管理プロセスへの適用

同様に，図 16 の (2)在庫待ち解消プロセスの下位レイヤからサービスを抽出した．図 18 に提案手法の適用結果を示す．出庫依頼プロセスは，(1)入庫依頼から必要なパラメータを切り出す，(2)切り出したパラメータの正当性チェックを行う，(3)チェックの結果問題がなければ酒在庫データに出庫依頼を受けた酒があるかチェックする，(4)酒があれば酒在庫データを更新して更新したデータについて出庫通知を出力する，という四つのプロセスから構成されている．(1)，(2)，(3)，(4)のプロセスに対して，(ルール 1)，(ルール 4)を適用すると，(1)＋(2)｜(1)＋(3)｜(1)＋(4)の 3 つの結合プロセスがサービスとして抽出された．よって，出庫管理プロセスは入力チェックサービス，酒在庫チェックサービス，酒在庫更新サービスの 3 つに分割することが可能である．

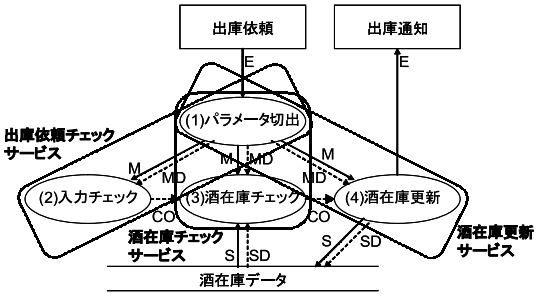

図 18　出庫依頼プロセスから抽出可能なサービス

5. 提案手法の特徴

5.1 サービス内部仕様の隠蔽

提案手法により抽出されるサービスでは，サービスを構成するプロセスの実行ロジックや，実装に依存した内部データといった内部仕様が隠蔽される．表 1 はケーススタディにおいて酒在庫管理システムから抽出したサービスの一覧である．レイヤ 1 の入庫管理サービス，出庫管理サービスは，4.2 節にて抽出したサービスである．レイヤ 2 のサービスはいずれも 4.3 節，4.4 節にて抽出したサービスである．なお，レイヤ 0 の酒在庫管理サービスは，入庫管理サービスと出庫管理サービスを統合したものであり，酒在庫管理システムから抽出可能なサービスの中では最も粗粒度である．

提案するサービス抽出ルールによって，決められた順序で実行しなければならないプロセスは同一のサービス内に隠蔽され，その実行順序をサービス利用者が意識する必要はない．抽出したサービスの候補は，単独で実行することも，繰り返し実行することも，他のサービスと組み合わせて実行することも可能である．表 1 に示したサービスの候補を例にとると，入庫管理サービスと出庫管理サービスは単体で実行することも可能であるし，組み合わせて実行することも可能である．よって，サービスの条件 1 を満たす．なお，酒の在庫が不足している状態で出庫管理サービスを実行した場合には，在庫不足によりエラーの応答が返されることになる．

また，提案手法により抽出したサービスの候補は，外部データを入力データとするか，入力データが不要である．システムの実装に強く依存するモジュールデータ，システムデータともにサービス利用者は意識する必要はない．表 1 に示したサービスの候補では，いずれのサービスも入出力データは外部データまたはデータ不要となっており，システムの実装に依存するモジュールデータおよびシステムデータを隠蔽している．よって，サービスの条件 2 を満たす．なお，出力データについては，サービスを実行した結果出力される外部データだけでなく，サービスを実行した結果（正常終了または異常終了，チェックサービスであれば真偽も）を利用者に通知する必要があるだろう．

表 1　酒在庫管理システムから抽出したサービス一覧

レイヤ	サービス名	入力	出力
0	酒在庫管理	入庫依頼 出庫依頼	出庫通知(出庫の場合) サービス実行結果(成功/失敗)
1	入庫管理	入庫依頼	サービス実行結果(成功/失敗)
2	入庫依頼チェック	入庫依頼	サービス実行結果(真/偽/失敗/)
2	酒在庫蓄積	入庫依頼	サービス実行結果(成功/失敗)
1	出庫管理	出庫依頼	出庫通知 サービス実行結果(成功/失敗)
2	出庫依頼チェック	出庫依頼	サービス実行結果(真/偽/失敗/)
2	酒在庫チェック	出庫依頼	サービス実行結果(真/偽/失敗/)
2	酒在庫更新	出庫依頼	出庫通知 サービス実行結果(成功/失敗)

5.2 DFD の詳細度に応じたサービス抽出

提案手法では，ソースコードから得られた DFD に対して，任意の DFD レイヤを選択することができる．上位レイヤに対して適用した場合には抽出されるサービスの粒度は粗くなる．この場合抽出されるサービスは，下位レイヤのサービスをまとめた高機能なものとなる．下位

レイヤに対して適用した場合には抽出されるサービスの粒度は細かくなり，低機能ではあるが再利用を想定したサービス抽出が期待できる．表 1 で示したサービスの候補を例にとると，最上位レイヤ（レイヤ 0)のサービスは酒倉庫管理サービスであり，入庫管理サービスや出庫管理サービスをまとめた高機能なサービスである．一方で，最下位レイヤ（レイヤ 2)のサービスとして酒在庫チェックサービスが抽出可能である．このサービスは出庫依頼に含まれる酒の在庫が十分にあるかどうかチェックしてサービス利用者に通知する．あらかじめ在庫があるかどうかチェックしたい場合などに利用可能であろう．このようにして，抽出するサービスの粒度を自由に決定することが可能である．従って，DFD の適度な詳細度を決定して提案手法を適用することにより，サービスの条件 3 を充足可能である．

5.3　レガシーソフトウェアの実態を考慮したサービス抽出

提案手法を用いることで，レガシーソフトウェアの実態を考慮したサービスの抽出が可能となる．2.3 節にて述べたように，SOA に基づくシステムを新規開発する手法では業務プロセスを元にサービスを設計する．ここでベースとなる業務プロセスはシステムの仕様を作成する際のインプットとなるものである．詳細は本稿では割愛させて頂くが，単一の酒在庫管理システムの仕様を元に複数のプログラムを作成した場合に，作成者により実装が異なる結果となった．当然，それぞれのプログラムから抽出可能なサービス候補も別のものとなった．このため，仕様よりもさらに上位の行程で作成される業務プロセスを元にサービスを設計した場合，システムの実装と合わないサービスの候補ができあがってしまう可能性はさらに高くなるといえるだろう．しかしながら，提案手法ではソースコードを入力としてサービスを抽出しているため，システムの実装を正確に反映することができる．

5.4　機械的なサービス抽出

提案手法を用いることで，レガシーソフトウェアから機械的にサービスを抽出することができる．提案手法では，プログラムコードに含まれる処理の依存関係に着目してサービスを抽出している．そのため，レガシーソフトウェアのプログラムコードが理解できていれば良く，プログラムコード作成のインプットとなる仕様書類の理解は不要である．これにより，プログラムの各処理の根底にある思想を理解していない場合であっても，機械的にサービスの抽出を行うことが可能となっている．

しかしながら，ソースコードのみを情報源として各プロセスの詳細な意味を理解することは難しい．ビジネスやプロセスの意味解析にまで踏み込んでいないため，抽出されるサービスはあくまでサービスの必要条件を満たすサービスの候補である．サービスの粒度は一般的にサブシステム単位程度の粗さであるべきとされており[10]，この粒度を実現するためには，提案手法により抽出したサービスの候補の粒度は必ずしも最適とはいえず，抽出したサービスを適切に組み合わせる必要がある．サービスの候補のより正確な絞込みについては今後の課題としたい．

6.　お　わ　り　に

本稿では，サービス指向アーキテクチャの構成要素であるサービスの条件として，自己完結，オープンなインタフェース，任意の粒度の三つを明確にした．その上で，レガシーソフトウェアのソースコードを DFD に変換し，データ依存の解析を行うことで，サービスの 3 条件を満

たす形でレガシーソフトウェアからサービスの候補を抽出する手法を提案した．また，提案手法に基づき，酒在庫管理システムの複数の実装からサービスの候補を抽出し，抽出したサービスの候補がサービスの 3 条件を満たしていることを確認した．

今後は，実際に業務で利用されているレガシーソフトウェアに提案手法を適用することを考えている．本稿のケーススタディでは酒在庫管理システムの実装に提案手法を適用したが，現実に稼働している大規模なソフトウェアには適用していない．また，本稿では C 言語で記述されたプログラムに対して提案手法を適用したが，COBOL で記述されたプログラムについても評価が必要であろう．今後は実際に業務で利用されているレガシーソフトウェア，特に COBOL で記述されたレガシーソフトウェアに提案手法を適用し，提案手法の実用性の評価を行いたい．その際には，提案手法の自動化が必要となるであろう．本稿執筆時点ではサービスの抽出を手動で行ったが，大規模なソフトウェアに提案手法を適用する場合に，手作業によりサービスの抽出を行うのは負荷が大きい．ソースコードを入力としてサービスの候補を出力するツールが必要になるものと思われる．また，提案手法により抽出されるのは，サービスの必要条件を満たすサービスの候補である．サービスの十分条件の判定を行い，最適粒度のサービスを抽出する手法を考えていきたい．

謝　辞　本稿の内容は，奈良先端科学技術大学院大学情報科学研究科在籍中に研究したものであり，松本健一教授をはじめとして，門田暁人助教授，中村匡秀助手，大平雅雄助手，ソフトウェア工学研究室の方々の多大なる御指導，御支援の賜物である．ここに深く感謝の意を表するものである．

参考文献

［1］P. Benedusi, A. Cimitile, and U. De Carlini,“ A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance, ” In Proceedings of International Conference on Software Maintenance（ICSM'89）, pp.180–189, October 1989.
［2］H. He,“ What is Service-Oriented Architecture？, ” http：//webservices.xml.com/pub/a/ws/2003/09/30/soa.html
［3］牧野友紀,“ ビジネス環境と実装システムを繋ぐ BPM と SOA ”, 情報処理，Vol.46, No.1, pp.60–63, January 2005.
［4］松村知子，門田暁人，松本健一,“ 潜在コーディング規則違反を原因とするフォールトの検出支援方法の提案 ”, 情報処理学会論文誌, Vol.44, No.4, pp.1070–1082, April 2003.
［5］門田暁人，佐藤慎一，神谷年洋，松本健一,“ コードクローンに基づくレガシーソフトウェアの品質の分析 ”, 情報処理学会論文誌，Vol.44, No.8, pp.2178—2188, August 2003.
［6］A. B. O ’Hare, E. W. Troan,“ RE-Analyzer： From source code to structured analysis, ” IBM Systems Journals, Vol.33, No.1, 1994.
［7］M. P. Papazoglow, D. Georgakopoulos,“ Service Oriented Computing, ” In Communications of the ACM, Vol.46, No.10, pp.25–28, October 2003.
［8］城田真琴,“ IT ソリューションフロンティア：技術 SOA–サービス指向アーキテクチャー ”, http：//www.itmedia.co.jp/survey/articles/0502/25/news 001.html
［9］P. T. Ward,“ The Transformation Schema： An Extension of the Data Flow Diagram to Represent Control and Timing, ” IEEE Trans. on Software Eng., Vol.12, pp.198–210, 1986.
［10］山岸重雄,“ .NET の本来の姿と現状，そして日本ユニシスの.NET ”, UNISYS 技報 85 号，Vol.25 ,No.1 ,May 2005.
［11］山田正隆，陸振宏,“ XML Web サービスの技術動向 ”, 東芝レビュー，Vol.58, No.2,

pp.3–6, 2003.
[12] 山崎利治, “ 共通問題によるプログラム設計技法解説 ”, 情報処理, Vol.25, No.9, pp.934–935, 1984.
[13] S.W.L. Yip, “ Software Maintenance in Hong Kong, ” In Proceedings of the International Conference on Software Maintenance（ICSM ’ 95）, pp.88–97, May 1995.
[14] 吉松史彰, “ XML Web サービスにおける疎結合とは何か ”, http://www.atmarkit.co.jp/fdotnet/opinion/yoshimatsu/onepoint03.html

執筆者紹介　木　村　隆　洋（Takahiro Kimura）
1999 年日本ユニシス(株)入社．UNIX 上での金融外接系パッケージ，Windows 上での金融外接系パッケージの開発，保守に従事．2004 年 4 月より 2006 年 3 月まで奈良先端科学技術大学院大学に留学．現在，日本ユニシス・ソリューション(株)ファイナンシャルシステム開発本部ミドルウェア開発部基盤ビジネス室に所属．