情報処理学会論文誌 **Vol.53 No.2 683–697 (Feb. 2012)**

メタデータ制約を用いた協調プロトコルの自動合成手法

髙橋 竜一[1,a) 石川 冬樹[2] 本位田 真一[2,3] 深澤 良彰[4]

受付日 2011年5月30日, 採録日 2011年11月7日

概要：SOC（Service Oriented Computing）のような複数の主体が連携し，協調動作をするシステムの構築では協調プロトコルの設計が重要である．しかし，セキュアなアプリケーション設計に必要なロギングや認証処理等を組み込むと，協調プロトコルは肥大化し，設計のコストは増大する．大規模な協調プロトコルを設計する手法として，協調プロトコル合成があるが，既存の協調プロトコル合成手法は，合成順序への依存が大きく，目的の協調プロトコルを得られないときは複数回の試行を必要とした．また，合成対象が協調プロトコル内の正常系処理のみであり，例外処理や終了処理の合成が扱えなかった．本論文では協調プロトコルの自動合成手法を提案する．自動合成手法では合成者が合成順序を考慮する必要がなく，合成器が指示した性質を満たす合成方法を探索する．また，例外処理や終了処理の合成も扱えるようにし，より柔軟な協調プロトコルを容易に合成する手法を提案する．

キーワード：サービス指向コンピューティング，相互作用設計，協調プロトコル合成

Automatic Coordination Protocol Composition Approach Using Metadata-base Constraints

Ryuichi Takahashi[1,a) Fuyuki Ishikawa[2] Shinichi Honiden[2,3]
Yoshiaki Fukazawa[4]

Received: May 30, 2011, Accepted: November 7, 2011

Abstract: Specifying coordination protocols are vital in systems which need to collaborate several entities. To implement secure application, it is necessary to build in loggings, authentications and so on. But they expand coordination protocols. Coordination protocol composition approaches reduce the costs of specifying such a large coordination protocols. In existing composition approaches, composition results depend on orders of compositions. And existing composition approaches can not treat exception processes and finalization processes. A new approach is proposed that search optimal compositions automatically and treat exception process and finalization processes.

Keywords: service oriented computing, interaction design, coordination protocol composition

1. はじめに

近年，SOC（Service Oriented Computing）のように，複数の主体が，独自の専門を活かし，独立性や再利用性を確保しつつサービスを構築し，それらを組み合わせることでアプリケーションを構築する手法が一般的となってきている．SOCのように複数の主体が連係するアプリケーションの構築には，全体の協調動作を設計段階で最初に決める必要がある．最初に全体の協調動作を決めることによって，各主体が連携するのに必要な手順やインタフェースが決ま

[1] 早稲田大学メディアネットワークセンター
Media Network Center, Waseda University, Shinjuku, Tokyo 169–0071, Japan
[2] 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[3] 東京大学
The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[4] 早稲田大学
Waseda University, Shinjuku, Tokyo 169–8555, Japan
[a) ryuichi-t@aoni.waseda.jp

© 2012 Information Processing Society of Japan

り，主体ごとの開発ができるようになる．設計段階において全体の協調動作を定義したものを協調プロトコルと呼ぶ．

協調プロトコルはアプリケーションの協調動作の部分だけを抽出したものである．構成要素は協調主体であるロールや主体間で送受信されるメッセージ，メッセージによってやりとりされる変数がある．協調プロトコルは協調動作の部分のみを取り出したものなので，実装方法や内部動作等は扱わない．協調プロトコルの設計はロール数やメッセージ数に依存し，プロトコルの規模が大きくなるほど複雑になる．特に，ロギングや認証，課金等の処理はアプリケーションのメインロジックではないが，セキュアなアプリケーションを構築するには必要不可欠な機能であり，これらの機能をアプリケーションに組み込むと，必然的に協調プロトコルは複雑・大規模化し，その作成のコストは増大する．

セキュアなアプリケーションを構築するためのロギングや認証といった処理は，どんなアプリケーションにも登場する汎用的な処理でもある．既存の処理を再利用し，容易にメインロジックに組み込むことができれば協調プロトコル構築のコストは低減する．既存の協調プロトコルを再利用して目的の協調プロトコルを構築する手法に，協調プロトコル合成 [1], [2], [3] がある．プロトコル合成は，メインロジックや課金，認証といったプロトコルをそれぞれ個別の部品として扱う．プロトコル設計者は，目的の協調動作をする協調プロトコルを作るために，それらの部品をどのように組み合わせるかを合成記述という形で指定する．合成器に部品となるプロトコルと合成記述を与えることで，合成器は指示どおりにプロトコルを組み合わせて，目的の協調プロトコルを出力する．

筆者らは過去に協調プロトコル合成手法の 1 つとして，メタデータを用いた協調プロトコル合成手法 [4] を提案し，合成指示の抽象化を実現した．しかし，この手法は以下の点がまだ不十分である．

(1) 合成プロトコルが合成順序に対して依存

(2) 例外処理や終了処理の扱いの不十分さ

1 つ目の「合成プロトコルが合成順序に対して依存」とは，複数の合成を適用する場合に発生する問題である．従来のプロトコル合成手法だと，合成者が合成指示を順番に与え，各合成を段階的に行う形になる．したがって，合成する部品の順番で最終的にできあがるプロトコルが変わる．この方法では，先に適用された合成によって付与されたり変化したりした性質が，後からの合成によって崩されてしまう可能性がある．このため，プロトコル作成者は最終的にできあがった合成プロトコルが意図した性質を持っているかを検証し，場合によっては合成順番を変更したうえで再合成しなければならなくなる．

2 つ目は，合成の柔軟性に関する問題である．プロトコル合成によって様々な処理を組み合わせることが可能になる．しかし，プロトコル合成では部品プロトコルの処理はそのままの形で組み合わされ，挿入対象となったプロトコルの通常系列の一部として実行される．したがって，挿入された部品の実行が終わると，挿入対象のプロトコルに処理が戻ることになる．正常系列の処理として部品プロトコルが実行されるだけならこれで問題はないが，実際の処理には例外処理や終了処理といった特別な処理が含まれる．従来手法で例外処理を含む部品を合成した場合，例外発生時には合成によって追加された各部品の中で例外処理が実行されるだけである．合成プロトコルは単機能の部品プロトコルの組合せによって構成されているので，例外の影響は各機能の範囲でしか捕捉（catch）されない．しかし実際は，例外の影響は 1 つの部品の中で閉じるのではなく，場合によっては合成された協調プロトコル全体の観点で例外を捕捉する必要がある．このように合成で追加された処理における例外処理等が，他の協調プロトコルに対してどう影響するかを指定して合成することが必要になる．

本論文では，筆者らの過去の研究 [4] を拡張し，これらの問題の解決を目指す．これにより，従来の協調プロトコル合成手法の欠点を克服し，より効率的に目的の協調プロトコルを作成することのできる手法を提案する．

本論文の構成は以下のようになる．2 章で筆者らの過去の研究であるメタデータを使用した協調プロトコル合成手法について紹介する．3 章から 5 章でこの過去研究 [4] の拡張を行う．3 章では自動合成に必要な問題の解決を行う．4 章で自動合成プロセスを定義する．5 章では例外および終了処理の伝播を実現する．6 章ではケーススタディを用いて，本手法の合成を説明する．7 章では本手法の有効性等について考察する．8 章で関連研究を示し，9 章でまとめと今後の課題を述べる．

2. メタデータを使用した協調プロトコル合成

筆者らは文献 [4] で協調プロトコル合成にメタデータを導入した協調プロトコル合成手法を提案した．本手法では，各協調プロトコルは協調プロトコル記述言語である WS-CDL [5] によって定義され，各協調プロトコルを構成するメッセージや変数にはメタデータが付与される．メタデータは単語集合であり，FIPA-ACL [6] の Communicative Act 等を参考に，利用者間で語彙を共有することで，メッセージ等の意味を表現することができる．プロトコル開発者が，プロトコルを構成する各メッセージに対して，このメタデータを付与し，合成時にはこのメタデータを使用して合成指示を行う．また，プロトコル間での変数の共有判断もこのメタデータを用いて行う．

たとえば，**図 1** のように課金処理を合成によって付与する場合を考える．合成箇所はメッセージに対する相対位置という形で特定する．図 1 の場合は，「コンテンツ送受信の前に課金を行いたい」という合成意図がある．したがっ

© 2012 Information Processing Society of Japan

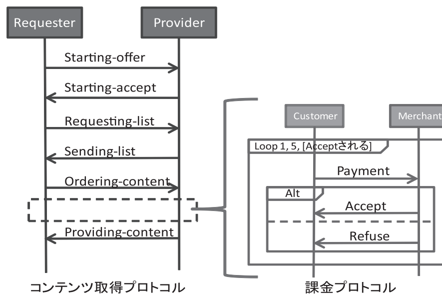

図 1 コンテンツ取得＋課金
Fig. 1 Composition of obtaining content and payment.

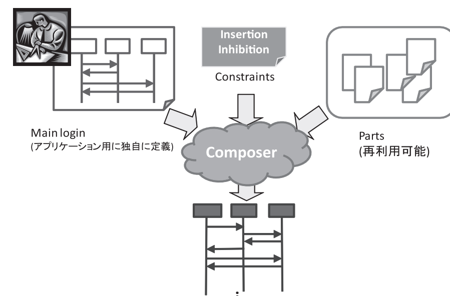

図 2 自動合成の合成プロセス
Fig. 2 Automatic composition process.

て,「コンテンツを送受信する」メッセージの「直前」が合成箇所となる.

従来手法では,この「コンテンツを送受信する」メッセージをメッセージ名を用いて指定してきた.具体的には“Providing-content”メッセージを直接指定する.この方法だと,合成者によって意識的に,「コンテンツ送受信」＝「“Providing-content”メッセージ」という変換を行うことが必要になる.変換には合成に使用する部品プロトコルの十分な理解が必要となる.

メタデータを使用することでより直接的な合成意図の表現が可能になる.図 1 の“Providing-content”メッセージには,“content, inform”といったメタデータが与えられる.これは,“content: コンテンツに関するメッセージ”,“inform: 何らかのデータを通知(受け渡す)ことが目的のメッセージ”といったことを表すメタデータである.合成指示にはこのメタデータを直接指定し,“inform, content”を持つメッセージの“後”といった形で指定する.メタデータはメッセージの意味と直結しているため,メッセージ名を使用するよりもより意味に沿った合成指示を行うことができる.また,メタデータは複数のメッセージが共通するものを持つことができる.したがって,複数の性質(メッセージ)に横断的な合成を 1 度に指定することができる.本手法によって,協調プロトコルを意味に沿って組み合わせて,複雑なアプリケーションの協調動作を開発することができるようになった.

3. 自動合成

文献 [4] で提案した合成手法により,合成指示をより抽象化し,意味に沿った合成を表現できるようになった.しかし,この手法はメインとなる協調プロトコルに対して,1 つのサブとなる協調プロトコルを合成する単機能合成方法であり,合成を段階的に繰り返すことで,目的の合成プロトコルを得る.この手法では,合成を適用する順番によって,完成する合成プロトコルが異なり,1 章で示した,「合成プロトコルが合成順序に対して依存」の問題が残されていた.そのため,合成後の性質検証が必要であり,検証結果によっては,合成順序等を見直し,再合成を行うといった試行コストが開発者の負担になっていた.合成者が手動で合成を試行錯誤する必要があるこの合成方法を,便宜上,手動合成と呼ぶ.

本論文では,この手動合成による開発者の試行コストの軽減を 1 つの目的とする合成方法を提案する.この合成方法を手動合成に対して自動合成と呼ぶ.提案する自動合成とは,**図 2** に示す合成プロセスをとる.自動合成プロセスでは,合成の順番を開発者(合成者)は考慮せず,メインロジックの協調プロトコルと,汎用部品の部品プロトコルをまとめて 1 度に合成器に入力する.また,自動合成では合成制約を部品プロトコルとともに合成器に入力する.合成制約は各部品プロトコルの合成の可否に関する制約を記述したもので,合成プロトコルは最終的にこの制約を満たしている必要がある.合成器は,入力された部品プロトコルを組み合わせる際に,すべての制約を満たす組み合わせ方を探索し,合成プロトコルを作成・出力する.

本章では自動合成実現に必要な問題の解決を行う.

3.1 メタデータを用いた合成制約記述

合成制約では各部品プロトコルの合成の可否を指定する.各部品プロトコルはそれぞれ汎用的な機能を担っており,各機能がどのようなタイミングで実行されてほしいか,もしくは実行してほしくないかを合成制約という形で表現し,合成箇所の判断に利用する.合成制約はメタデータを使用して記述する.メタデータを使用した合成は,変数名やメッセージ名を使用する場合よりも,より抽象的に合成意図を表現でき,かつ各部品プロトコルに横断的な性質に対する同一の合成を 1 度に指定することが可能というメリットがある.

制約の表現は次の書式を用いて行う.

合成許可制約

Insertion(*PartsProtocol, Metadata1, [Metadata2,] RelativePosition, ExceptionType, Times*)

合成禁止制約

Inhibition(*PartsProtocol, Metadata1, [Metadata2,] RelativePosition,Times*)

合成制約は,*PartsProtocol* 要素で指定した各部品プロ

© 2012 Information Processing Society of Japan

表 1　*RelativePosition* 要素で指定される値
Table 1　Values specified for *RelativePosition* elements.

値	意味
Before	メッセージよりも前
ImmediateBefore	メッセージの直前
After	メッセージより後
ImmediateAfter	メッセージの直後
Between	2 つのメッセージの間

表 2　*Times* 要素で指定される値
Table 2　Values specified for *Times* elements.

値	意味
All	一致するメッセージすべて
FirstOne	一致する中で最初のメッセージ
LastOne	一致する中で最後のメッセージ
First_First	最初の m_1 から最初の m_2 まで
First_Last	最初の m_1 から最後の m_2 まで
Last_First	最後の m_1 から最初の m_2 まで
Last_Last	最後の m_1 から最後の m_2 まで

トコルに対して，上記の書式で合成許可または合成禁止を記述する．部品プロトコルの挿入箇所は，メタデータによりメッセージを特定したうえで，その前後または間という形で指定する．*Metadata1* 要素（および *Metadata2* 要素）にメッセージを特定するためのメタデータを指定する．*RelativePosition* 要素は特定したメッセージに対して，どの位置にプロトコルの挿入を許可または禁止するかの指定に用いる．*RelativePosition* 要素は 5 種類の値をとる．**表 1** に各値と，その値が指す位置・範囲を示す．*RelativePosition* 要素に *Between* を指定したときにのみ *Metadata2* 要素を指定することができ，*Metadata1* 要素によって特定されたメッセージから，*Metadata2* 要素によって特定されたメッセージの間が対象範囲となる．*ExceptionType* 要素は *Propagation* か *Individual* の値をとり，その部品プロトコル中で例外が起こった際の処理方法を指定する．例外処理方法の詳細については 5 章に示す．*Times* 要素は，**表 2** に示す 7 種類の値をとり，指定したメタデータにマッチするメッセージが複数存在したときの選択方法の指定に使用する．*Times* 要素は，*RelateivePosition* 要素が *Between* かそれ以外かによって，指定できる値が変わる．*RelativePosition* 要素が *Between* の場合は，*Fisrt_First*，*First_Last*，*Last_First*，*Last_Last* の 4 つの値をとる．指定したメタデータのうち，*Metadata1* にマッチするメッセージを m_1，*Metadata2* にマッチするメッセージを m_2 と表現した場合，表 2 に示す範囲が，合成を許可または禁止する範囲となる．*RelativePosition* 要素が *Between* 以外の場合は，*All*，*FirstOne*，*LastOne* の 3 種類の値をとり，表 2 に示す実行順番に基づいたメッセージの選択を行う．

合成許可および合成禁止の制約は，合成者が意図する合成プロトコルを作成するために記述する．また，個々の部品プロトコル作成者が，合成によって崩されてほしくないメッセージの順番や性質がある場合に，合成後もそれらが維持されるよう，合成禁止制約を記述する場合もある．ほかにも，既存の実装済みのサービス等を部品プロトコルとしてプロトコル合成に使う場合に禁止制約を用いる．実装済みサービスはすでにメッセージの順序等が確定してしまっているため，それらの間に別の処理を挿入することはできない．*Between* の合成方法で，*Metadata1* にサービスの最初のメッセージ，*Metadata2* にサービスの最後のメッセージを指定し，*PartsProtocol* に “Any” と指定することで，既存サービスに他の処理の挿入を禁止することができ，メッセージ順序を保護することができる．

また，合成許可制約と合成禁止制約が示す範囲が競合した場合，合成禁止制約の方を優先する．

これらの形式に従い，合成する各協調プロトコルに対して合成許可および合成禁止制約を用意する．

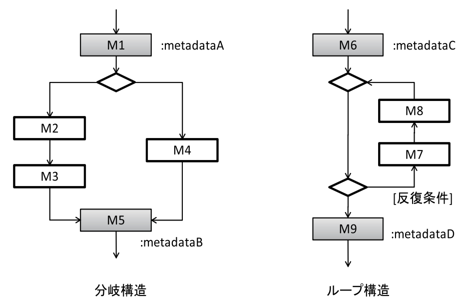

図 3　制御構造の例
Fig. 3　Example of branch and loop structures.

3.2　制御構造を考慮した合成範囲修正

本手法で扱う協調プロトコルは WS-CDL で記述されている．WS-CDL は制御構造（分岐構造やループ構造）を表現することができ，部品プロトコルも当然それらの制御構造を持つものが使用されうる．部品プロトコル内の制御構造箇所に対して合成を実行する場合，単純にメタデータにマッチした部分を合成箇所とすると，制約を満たすことができない場合が発生する．

たとえば，**図 3** の左図のような分岐構造を考えてみる．合成許可制約で metadataA と metadataB の間に挿入することを考える．この例では，メッセージ M1 とメッセージ M5 の間のどこかに挿入することが必要である．ここで，M1 の直後や M5 の直前といった分岐構造の外部に挿入を実行する場合，実行時に分岐構造のどちらのパスを通っても制約を満たす．しかし，M2，M3，M4 の前後といった分岐構造の内部の 1 カ所に挿入を実行した場合，問題が生じる．完成した合成プロトコルを実行した際の状況によっては，制約が満たされないパスを通ってしまう可能性があり，必ず制約を満たすことを保証することができなくなる．

同様にループ構造の場合，ループ制御の条件によって

© 2012 Information Processing Society of Japan

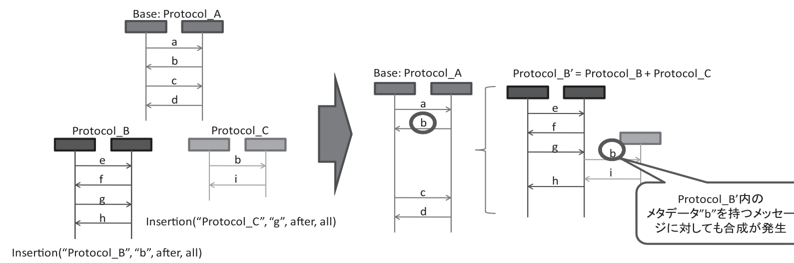

図 4 問題：循環合成の例
Fig. 4 Example of recursive compositions.

は，ループ構造内部のメッセージが 1 度も実行されない場合がある．図 3 の右図のループ構造に対して，分岐構造と同様に挿入箇所候補が範囲となる合成許可制約の適用を考える．metadataC から metadataD の間への挿入を例として考える．条件分岐のときと同様に，M7 や M8 の前後といった，ループ構造の内部に挿入を実行した場合，ループ構造内の処理が実行されないと，制約を満たすことができなくなってしまう．

このように，合成箇所の候補となる範囲に制御構造が含まれている場合，実行時にどのような実行パスを経ても制約が満たされるよう，合成を工夫しなければならない．始点と終点からなる範囲が形成されるとき，始点を通過後に必ず終点を通過するためには，始点と終点が同じスコープにある必要がある．したがって，複数のスコープにまたがる範囲は，修正する必要がある．以下に，各場合における，合成候補範囲の修正方法を示す．

3.2.1 制御構造共通のルール

制御構造の共通ルールとして，合成候補範囲が基点メッセージのスコープより上位のスコープにまたがる場合，上位のスコープは合成候補範囲から外す．ここで修正対象としているのは，あくまで合成箇所の候補となる範囲であり，最終的に判断される，制約が作り出す範囲とは異なる．これによって，分岐構造やループ構造が構成するスコープの 1 つに基点メッセージが含まれているとき，合成候補範囲はそのスコープ内に閉じることになる．*Between* の合成では，基点が始点と終点の 2 つになるので，下位の方のスコープに合成範囲を合わせる．たとえば，範囲の始点となるメッセージが分岐構造の外，範囲の終点メッセージが分岐したパスのうちの 1 つである場合，分岐構造の外は合成候補範囲から外す．合成候補範囲は，分岐したパスの最初から終点メッセージまでになる．

3.2.2 分岐構造

分岐構造は，合成の基点となるメッセージが分岐構造外にあるとき，分岐したパスのいずれを通っても制約が満たされるようにすることが必要となる．そこで制御構造のスコープを考慮し，基点となるメッセージよりも下位のスコープの分岐構造に挿入を行う場合，その分岐構造の別条件パスのすべてにも同じプロトコルを合成する．

また，*FirstOne* や *LastOne* のように，条件に一致する 1 つの基点メッセージに対して合成をする場合，そのメッセージが分岐構造の分岐パスの中に含まれている場合は，並列する他の分岐パスもチェックし，同じメタデータを持つメッセージがあれば，それに対しても同様に合成許可制約を適用する．

3.2.3 ループ構造

ループ構造の場合，ループ部分に合成を実行すると，ループ回数が 0 のときに制約が満たされなくなってしまう．そこで，基点メッセージの下位のスコープのループ構造に挿入範囲がまたがったとしても，そのループ構造内には合成を行わないこととする．これにより，ループ回数に影響を受けず，制約が満たされるようになる．

3.3 モデル合成

自動合成をするにあたり，WS-CDL の協調プロトコル上で直接，合成箇所を探索しながら，挿入を実行すると「合成循環」と「合成箇所の競合」という問題がおきる．

合成循環とは，合成が永久に収束しない問題である．プロトコル合成では合成によって追加された処理の分，新たなメタデータが増える．したがって，増えた分のメタデータ対して追加の合成が発生することがある．特に *All* によって，制約を満たすすべての場所に対して挿入を許可する場合，合成が新たな合成を呼び，永久に収束しない場合がある．

図 4 に例を示す．この例はメインロジックとして Protocol_A を，部品プロトコルとして Protocol_B と Protocol_C を合成する．まず，2 つの合成許可制約のうち，Protocol_C を合成するものを適用することを考える．この制約の適用によって，Protocol_B に Protocol_C が合成される．更新された協調プロトコルを Protocol_B' と表現する．次に更新された Protocol_B' を挿入するための制約を適用することを考える．この制約の適用によって，Protocol_A に Protocol_B' が挿入される．それと同時に，部品プロトコルの Protocol_C も基点となるメタデータを持つメッセージを持っているため，Protocol_B' に対しても Protocol_B' を挿入するという形になる．Protocol_B' の挿入によって，付与された部分にもメタデータ "b" を含むメッセージがあ

© 2012 Information Processing Society of Japan

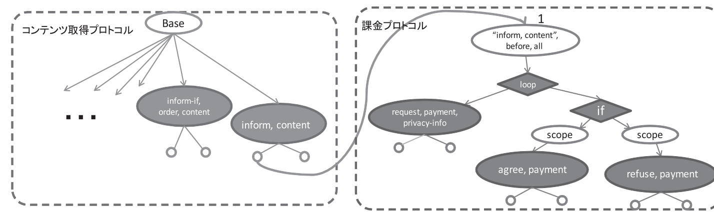

図 5　循環検証ツリー
Fig. 5　Detection trees for recursive compositions.

る．したがって，制約を満たすためにはこのメッセージに対してもさらに合成を適用しなければならない．このように，適用しなければならない合成が循環し，無限に合成が行われるようになってしまう．この問題を本論文では循環合成と呼ぶ．

合成箇所の競合は，まったく同じ箇所に複数の合成が同時に発生してしまうという問題である．たとえば，ある 2 つの部品プロトコルを，それぞれメッセージ M の *ImmediateAfter* に挿入する場合を考える．このとき，合成箇所がまったく同じであるために，後から実行した合成が，先に実行した合成で作成されたメッセージ順序を崩してしまい，メッセージ M の直後の位置ではなくなる．これは合成順番を逆にしても解決することはできない．

協調プロトコル上で，直接合成を行ってしまうとこの循環合成や合成箇所の競合が発生してしまうことがある．そこで，実際の合成を行う前に，協調プロトコルのモデル上で合成を検証し，これらの問題の発見を行う．合成モデルの検証には，各協調プロトコル間の干渉グラフを作成し，そのグラフを探索することで問題の発見を行う．

まず，合成に使用する協調プロトコルごとに，図 **5** のような木構造のモデルを作成する．このモデルは，ルートにその協調プロトコルの挿入許可制約を持ち，ルートノードの子に協調プロトコル中のメッセージを表すノード（メッセージノード）を持つ．メッセージノードは各メッセージのメタデータ情報を持ち，左から右にメッセージが実行される順番に並んでいる．各メッセージノードは 2 つの葉ノードを持ち，この葉ノードはそれぞれメッセージの直前および直後と，協調プロトコルが挿入される箇所を表している．

次に，各協調プロトコルの挿入許可制約を基に，制約を満たす箇所（葉ノードを選択し），ルートノードと葉ノードの間にエッジを作成する．また，このエッジにはたどることのできる回数の情報があり，合成許可の *Times* 要素が *All* のときは無限回，*Times* 要素が *FirstOne* または *LastOne* のときは 1 回となる．

最後に連結されたモデルを探索し，循環合成がないかを確認する．メインロジックである Base ツリーのルートから探索を始め，中間順探索にて各ノードを探索していく．このツリーを探索しきることができれば，循環合成は発生しない．探索の途中でループが検出された場合，その合成箇所に合成を行うと循環合成になってしまうので，合成方法を見直す必要がある．想定されるすべての連結方法でループが生成される場合は，その合成制約の組合せでは，目的の協調動作の合成を行うことができないので，合成制約自体を見直す必要がある．

また，同一の葉ノード（合成箇所）から複数の部品プロトコルにエッジが作成されている場合，これは合成箇所の競合を意味する．合成箇所の競合が発見された場合，モデルに従って実際に WS-CDL 上での合成を行うときに，競合のある部品プロトコルを並列構造をともなって合成する．並列構造にすることで，部品プロトコル間の順序の前後関係がなくなり，合成箇所の競合という問題を解決することができる．

このように実際の WS-CDL 上での合成の前に，まずモデル上で合成箇所を確認することにより，循環合成や合成箇所の競合といった問題を解決することができる．

3.4　制約検証

循環合成のチェックの後，作成された合成プロトコルがすべての制約を満たしているか検証を行う．この検証では，明示的な制約として合成者およびプロトコル開発者が与えた合成制約と，非明示的な制約として，変数制約の競合を検証する．

合成制約とは，合成許可制約や合成禁止制約のことである．変数制約とは，協調プロトコル中での変数および変数が持つデータに関する制約で，合成記述として合成者が与えるようなものではなく，協調プロトコル中に定義されている変数の性質のことである．協調プロトコルを定義する WS-CDL には変数に変更可能性を示す *mutable* 属性や *silent* 属性がある．*mutable* 属性が “false” となっていると，その変数は初期化後に書き換えを行うことができなくなる．協調プロトコル合成においてデータの受け渡しは，メタデータをもとにして変数の同一化をすることで実現するが，挿入する側と挿入される側の協調プロトコルで，これらの変数の性質が競合する場合がある．挿入される側で *mutable* 属性が “true” となっている変数，挿入する側で *mutable* が “false” となっている変数を同一化し，データの共有を行う場合，これはデータの書き換え可能区間中

© 2012 Information Processing Society of Japan

に一時的に書き換えを禁止する区間が挿入されることと同義となる．この場合は問題にならない．しかし逆に，挿入される側で *mutable* 属性が “false” となっている変数，挿入する側で *mutable* が “true” となっている変数を同一化して，データの共有を行う場合，変数の書き換えを禁止し，データの保護を行っているところに書き換えを許可する区間を挿入することとなる．この場合，本来持っていた特定区間内でのデータの保護という性質が挿入によって失われる可能性が発生してしまうため，制約違反となる．このように，変数が持つ制約についても検証を行う必要がある．

検証は分岐構造を考慮し，いずれの実行パスを通っても制約が満たされていることを確認する．具体的には，合成モデル上を中間順探査した際のメッセージの実行列を取得する．この際，分岐構造がある場合は，パス網羅をする形で，複数の実行列を取得する．ただし，ループ構造においては，ループ部分を 0 回実行した場合と，1 回のみ実行した場合の 2 種類だけを確認する．取得したすべての実行列において，各メッセージのメタデータと部品プロトコルの位置関係を確認し，すべての合成制約が満たされているか，また合成される側と合成する側の変数を確認し，変数制約の衝突が起こらないかを確認を行う．制約がすべて満たされていれば，その合成箇所で合成を行えば制約を満たした合成プロトコルが得られることが判定できる．

4. 合成アルゴリズム

本章では自動合成による合成アルゴリズムを示す．自動合成は 6 つのステップによって協調プロトコルを合成する．

Step1：部品プロトコルおよび合成制約の準備 合成に使用する部品プロトコルと，各部品プロトコルに対して，合成許可および合成禁止制約を用意する．

Step2：制約適用順序生成 合成許可制約を適用する順序を生成する．

Step3：合成 Step2 で生成した制約適用順序に従って，1 つずつメッセージベースの合成許可制約に従いモデル合成を行っていく．適用する合成許可制約のうち，*RelativePosition* 要素が，*Before*，*After*，*Between* の場合，合成箇所候補が範囲となるため，範囲の中から実際の挿入箇所を 1 つ選択する．また，各モデル合成を行うたびに，その合成が合成禁止制約に抵触していないかの評価も行う．抵触している場合は，その合成を実行せず，次の制約の適用に移る．すべての合成許可制約を適用するまで継続する．

Step4：循環検証 Step3 で生成された合成モデルを探索し，ループがないかを確認する．ループがあり，探索が終了しない場合は，循環合成が発生しているため，その合成箇所に対して合成をすることはできない．

Step5：制約検証 合成制約は適用した段階では満たされていても，他の制約の適用によって崩されてしまう場合がある．よって，得られた合成プロトコルに対して，すべての合成禁止制約の検証，および合成許可制約が満たされているか再度の検証を行う．また，同時に変数制約の競合を検証する．

Step6：合成 できあがった合成モデルに従って，実際に WS-CDL で記述された協調プロトコルを合成する．モデル上にマークされた箇所に部品プロトコルのメッセージ列を挿入し，新たな協調プロトコルとする．

Step4 で循環合成が発見された場合や，Step5 で制約に違反していることが判明した場合，Step3 にロールバックし，制約の合成箇所の候補範囲から，別の合成箇所を選び直す．Step3 の合成箇所候補の組合せをすべて網羅しても制約を満たさない場合は，Step2 へ戻り，制約適用順序を再生成する．制約適用順序および，合成箇所範囲をすべて網羅しても制約を満たすことができない場合，その制約の組合せを満たすことができない場合は，制約の組合せから見直す必要がある．以上のアルゴリズムを実行することで，制約を順守した合成プロトコルを得ることができる．

5. 例外伝播

3 章で述べた自動合成によって，アプリケーションのメインロジックに様々な処理を合成できるようになった．しかし，協調プロトコルは正常系の通常処理以外にも例外処理や終了処理から構成されている．協調プロトコルを記述する WS-CDL には，例外処理を定義する Exception Block や，終了処理を定義する Finalizer Block がある．たとえば，協調プロトコル内で例外処理が発生した場合，発生した例外は exceptionTypes という開発者が定義した名前で識別され，対応した Exception Block 内の例外処理が 1 つ実行される．例外処理実行後，その部品プロトコルは異常終了という形で終了し，親の処理に戻る．プロトコル合成において，各部品プロトコルはそれぞれが何らかの処理をアプリケーションに追加するものあり，汎用的な協調動作を定義したものである．したがって，そこに定義されている例外処理は自己の協調動作の範囲でしか責任を持たず，ともにどのような処理と合成されるかを想定したうえで定義されているとは限らない．また，実際に部品側で例外処理が発生した際に，その例外処理をどのように位置づけて，他の部品プロトコルの処理を終了するかまたは継続するかといった判断は合成のたびに変わりうる．

たとえば，部品として認証処理を合成する場合を考える（**図 6**）．この場合，認証失敗後に代替処理を実行することで処理の継続を許可する場合がある．図 6 の合成 1 のように，認証失敗時の例外処理内で代替処理が定義されていれば，その代替処理の実行後，元の正常系処理に戻すことで処理の継続ができる．ほかにも代替処理がない場合等は，認証失敗時には協調動作をその時点で終了することが考えられるが，その場合も 2 種類の終了方法が考えられる．1

© 2012 Information Processing Society of Japan

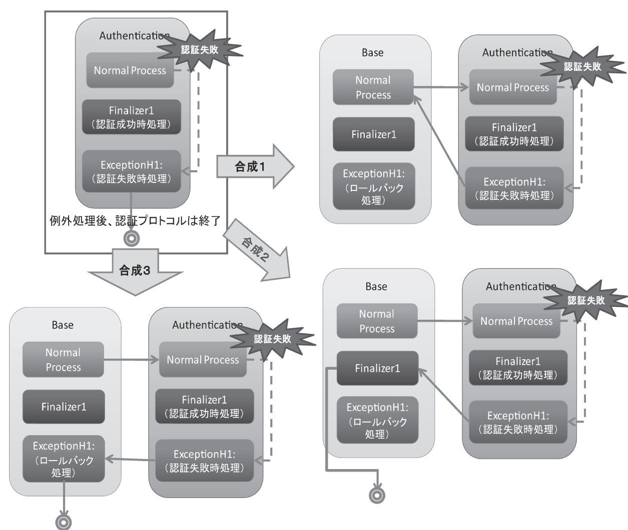

図 6　例外伝播の例
Fig. 6　Examples of exception propagations.

つ目は合成 2 のように，元の協調プロトコルの Finalizer へ伝播させることで，認証処理は失敗したが，全体としては正常終了という形で終了させる方法である．2 つ目は合成 3 のように，元の協調プロトコルの例外処理へ伝播させ，そこでロールバック処理等を実行したうえで異常終了という形で終了させる方法が考えられる．このように，例外処理は，部品プロトコルの設計者だけでなく，合成者によっても扱い方が変わる．

認証処理のような部品プロトコルは，汎用性を持たせるためにも例外処理が発生した後どうするかまでは責任を持つべきではなく，合成の際に，例外処理を実行した後どうするかを指定できる方がよい．したがって，各協調プロトコル間で例外処理を任意の処理に伝播させる仕組みが必要になる．これは，正常系の終了処理を行う Finalizer Block でも同様である．WS-CDL における Finalzer は実行後，その Finalizer Block を持つプロトコルは正常終了と見なされ，影響は部品プロトコルの中で閉じてしまう．終了処理を伝播させる仕組みはないため，親の協調プロトコルへの伝播を指定できるようにすることが必要である．そこで，例外処理や終了処理を他の例外処理，終了処理または正常系の処理に伝播させる仕組みを提案する．

5.1　例外処理・終了処理の伝播パターン

まず，例外処理（Exception Block）と終了処理（Finalizer Block）の伝播パターンを考える．協調プロトコルは，大きく分けて通常処理，例外処理，終了処理の 3 つに分けることができる．したがって，子プロトコルにおいて例外処理を実行した場合，親プロトコルを構成するそれら 3 種類の処理に，制御を戻すことが考えられる．同様に，終了処理も親プロトコルの 3 種類の処理，それぞれに伝播しうる．よって例外および終了処理の伝播では，それぞれ各 3 パターンの伝播が実行できるようにする必要がある．

5.2　例外処理・終了処理のメタデータ

次に，例外処理および終了処理の伝播の指定方法を定義する．WS-CDL では例外処理や終了処理の実行後は，親プロトコルの正常系処理に戻るため，親プロトコルの正常系への伝播は，特に指定する必要はない．それ以外の例外処理および終了処理の伝播は，子プロトコルから親プロトコルへの伝播，または子プロトコルからそれ自身の正常系への伝播となる．まず，伝播先が例外処理や終了処理となる場合，1 つの協調プロトコル中の複数の Exception Block や Finalizer Block のうち，どの Exception Block または Finalizer Block に伝播させるかの識別・指定が必要となる．合成段階ではどのプロトコルが親子関係になるかが分からず，したがって，例外処理や終了処理を指定するのに各 Block 名を用いることはできない．そこで，伝播する処理の指定にもメタデータを使用する．

Exception Block や Finalizer Block に付けられるメタデータを考える．例外処理を種類分けする 1 つの観点として，例外処理の目的があげられる．たとえば，何らかの Exception に対して例外処理を実行する場合，例外処理の目的を次のように分類することができる．

Altenative　実行を失敗した処理に代わる代替処理を行う．

Recovery　例外やエラーの修正を試みる．

RollBack　対象協調プロトコルを実行する前状態に戻す．

Abort　例外やエラーの修正を諦め，ただちに終了するための処理を行う．

各 Exception Block にはこれらの例外目的をメタデータとして付与し，例外伝播の際の例外処理の意図を表現することで，伝播対象を指定できるようにする．

同様に終了処理を分類する場合を考えるが，終了処理はすべて協調プロトコル（協調動作）を正常終了するための処理である．1 つの協調プロトコル中に複数の処理を定義する場合があり，その切替えは，アプリケーションレベルの条件判断に基づいて行われる．たとえば，購入処理において，金銭の支払い方法がクレジットカードや銀行振り込みのように複数から選べる場合，終了処理はこの支払方法の選択に応じて，切り替えられることが考えれる．1 つの協調プロトコル中で複数の終了処理が定義されている場合，終了処理の判別にはアプリケーションの知識が必要になるため，例外処理のように汎用的な種類分けをし，メタデータを定義することができない．したがって，終了処理の場合は，アプリケーションの知識を持ってメタデータの語彙の定義と共有をし，指定する必要がある．

5.3　伝播の指定

メタデータを用いて，例外処理と終了処理の伝播を指定

© 2012 Information Processing Society of Japan

する．伝播は以下の形で指定する．

伝播の書式

Propagation(*PartsProtocol, BlockName, Metadata*)

伝播の指定には 3 種類の情報を指定する．最初の 2 つ，*PartsProtocol* と *BlockName* は伝播元の特定に用いる．*PartsProtocol* には伝播元となる協調プロトコル名を，*BlockName* にはその協調プロトコルの Exception Block または Finalizer Block の名前が指定される．3 つ目の要素であるメタデータは伝播先の特定に用い，伝播元の協調プロトコルの親プロトコル内で，*Metadata* に指定したメタデータを持つ Exception Block または FinalizerBlock へ伝播する．ただし，通常処理の合成と異なり，伝播先は一意に特定できなければならない．

5.4 伝播の実行

本研究で協調プロトコルの記述方法として用いている WS-CDL では，どんな種類の例外であっても，例外処理終了後にはその協調プロトコルは終了となる．Perform アクティビティを実行して任意の部品プロトコルを呼び出している場合は，例外処理実行後は必ず呼び出し元の協調プロトコルへ実行が戻る仕様になっている．つまり WS-CDL において，例外は種類や影響の大きさ等に区別はなく，すべての例外を同様に扱い，いったん必ず終了するというセマンティクスを持っている．また，WS-CDL はプロトコル開発者が任意のタイミングで例外を throw することができない．したがって，通常は例外処理を連鎖的に実行させることができない．

このように，WS-CDL の仕様では，各協調プロトコル内の個々の例外処理単位で細かな制御をすることができない．そこで，本手法では部品プロトコルの単位で例外伝播の方針を指定できるようにする．例外伝播の方針は，合成許可制約の *ExceptionType* の値によって合成方法を変更することで，疑似的に例外伝播の仕組みを実現する．

ExceptionType が *Individual* を指定したとき，これは伝播を行わない従来の例外または終了処理を行うことを指定する．発生した例外の影響が小さい，または部品プロトコル内で影響を十分に吸収できるため，上位のプロトコルの例外処理や終了処理に伝播を行う必要がない場合にこの指定を用いる．つまり，伝播先が正常処理の場合がこの伝播になる．*Individual* が指定された場合，部品プロトコルの合成は Perform アクティビティを用いて挿入を行う．この方法では，部品プロトコルのコンポーネント構造が維持されたまま合成することが可能になる．

ExceptionType に *Propagation* が指定された場合，これは本章の目的である伝播をともなった例外および終了処理を行うことを指定する．発生する例外の影響が大きい，または部品プロトコル内だけでなく上位の協調プロトコルでも例外等の影響を考慮しなければならない場合にこの指定を用いる．上位のプロトコルの例外処理や終了処理へと伝播していくパターンである．この合成実現方法のイメージを図 7 に示す．*ExceptionType* で *Propagation* が指定された場合，挿入箇所に対して挿入をする部品プロトコルのメッセージ列をインライン展開することで合成を実現する．また，通常系列と同様に例外処理と終了処理がそれぞれ例外処理と終了処理のブロック内に追加される．しかし，そのままでは例外処理と終了処理が増えただけで伝播は実行されない．そこで，伝播指定の記述に従い，例外処理や終了処理の末尾に伝播先の例外処理や終了処理をコピーし，複数の例外処理や終了処理が疑似的に連鎖するようにする．このようにインライン展開することによって，元の部品プロトコル自体のコンポーネント構造が失われるが，影響を伝播させることが可能になる．

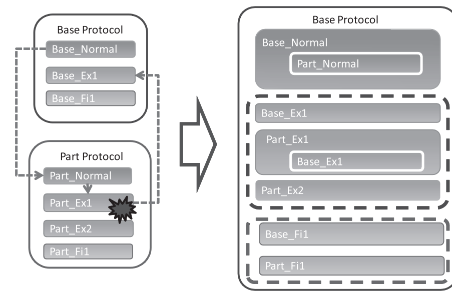

図 7 インライン展開による例外処理の合成
Fig. 7 Composition image by inline substitution.

このように合成方法を 2 種類に分け，それぞれ例外等の影響の大きさに応じて使い分けることによって，本来の WS-CDL の範囲で例外，終了処理の影響伝播を実行できるようにする．

6. ケーススタディ

本章では，実際に複数の協調プロトコルを合成し，本手法によって目的の合成プロトコルが得られることを示す．本章では，4 種類の協調プロトコルを合成する例を示す．合成に使用する協調プロトコルを図 **8** に示す．使用する 4 種類の協調プロトコルのうち，メインとなるのはコンテンツ取得プロトコルである．このプロトコルでは，Requester が要求したコンテンツを Provider が送信する．このコンテンツ取得プロトコルに対して，認証プロトコルと課金プロトコル，ロギングプロトコルを合成する．図 8 には各協調プロトコルのロールとメッセージが示されている．各メッセージ名の後ろの括弧内に示されているのが，そのメッセージに付与されているメタデータである．

表 **3** に合成に用いる各協調プロトコルへの合成制約を示す．コンテンツ取得プロトコルをベースプロトコルとして，認証，課金，ロギングの 3 種類のプロトコルに挿入許可制約を指定して合成を行う．各部品プロトコルは次のような意図で合成を行う．認証プロトコルは，Requeter が Provider

© 2012 Information Processing Society of Japan

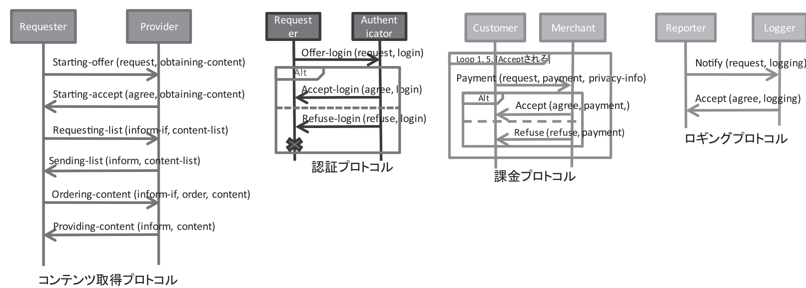

図 8　部品プロトコル
Fig. 8　Parts protocols.

表 3　合成制約
Table 3　Composition constraints.

部品プロトコル	制約番号	合成制約
コンテンツ取得		（なし）
認証	Ins_1	Insertion(“AuthenticationProtocol”, “inform-if”, Before, Propagation, FirstOne)
課金	Ins_2	Insertion(“PaymentProtocol”, “inform, content”, ImmediateBefore, Propagation, All)
ロギング	Ins_2	Insertion(“LoggingProtocol”, “send(Provider)”, ImmediateAfter, Individual, All)
	Inh_1	Inhibition(“LoggingProtocol”, “request, logging”, ImmediateAfter, Individual, All)

に対して最初に何らかのデータの提供を依頼する前に 1 度だけ認証が行われるように挿入を行う．具体的には，データの要求を意味する “inform-if” メタデータを持つメッセージの前に挿入を実行する．最初に 1 度だけ認証を実行するので，*Times* 要素は FirstOne を指定する．次に課金プロトコルは，コンテンツデータを Requeter に提供する直前に課金が実行されるように挿入する．挿入許可制約は，コンテンツデータの提供を意味する “content-send” メタデータの直前に挿入する．ロギングは，Provider がメッセージを送信するたびに記録を保存する方にする．したがって，Provider が送信したメッセージを表す “send(Provider)” メッセージの後にロギングを挿入する．

各制約を指定した後に，挿入制約の適用順序を作成する．まず，Ins_1，Ins_2，Ins_3 の順番で挿入許可制約の適用を考え，合成モデル上で各挿入許可制約に従いエッジを作成する．作成した合成モデル上で，挿入循環のチェックをするために，循環検証ツリーを作成する．作成した循環検証ツリーを**図 9** に示す．複数種類の協調プロトコルを合成するため，同じ種類の挿入が協調プロトコルに横断して発生しうるのが確認できる．この例の場合，認証プロトコルはコンテンツ取得プロトコル中の 2 カ所に，ロギングプロトコルは，コンテンツ取得プロトコルと課金プロトコルに横断する形で 5 カ所に連結されることとなる．認証プロトコルは *Times* 要素が *FirstOne* の指定になっているため，候補の 2 つのうち 1 つが選択され，エッジが作成される．図 9 の中で，コンテンツ取得プロトコルから認証プロトコルに伸びているエッジのうち，破線で表現されているが，選択されなかった方の合成を示すエッジである．もし，この合成箇所に従って合成を行った結果，認証プロトコルの合成が *FirstOne* の条件に合わない場所に合成されなかった場合，選択されなかったエッジから再選択をして合成を再試行することになる．また，禁止制約 Inh_1 によって，ロギングプロトコルの Notify メッセージ直後に，さらにロギングプロトコルが挿入されず，循環合成が回避されている．

ツリーの作成後，探索を行い循環が発生していないかを確認する．ベースであるコンテンツ取得プロトコルのルートより中間順探索で探索を行う．この制約および合成箇所の選択ではループが発生しないので，合成を実行することで合成プロトコルを得ることができる．

探索の際に得られたメッセージ列と合成制約を照会する．合成モデルよりパス網羅でメッセージの実行列を取りだす．すべての実行列において，メッセージの順序関係をチェックし，制約に違反していないことを確認する．本例では，図 9 の合成方法で循環や制約違反は発生しなかった．

最後に，実際に合成を行うことで目的の合成プロトコルを得ることができる．作成された合成プロトコルを**図 10** に示す．以上のように，ベースとなるコンテンツ取得プロトコルに対して，認証や課金といった処理を組み込んだ協調プロトコルを作成することができる．

また，合成プロトコルには，図 10 に示した正常系の協調動作のほかに，例外処理が含まれる．図 10 の課金処理部分は，支払いが成功する（Accept メッセージが送受信される）まで，ループ処理で繰り返し試行する設計になっているが，このループ処理には上限回数が定義されている（本例では最大 5 回としている）．したがって，5 回を超えて支払いが失敗した場合は，例外をスローし，課金処理を終了する．この例外は親のコンテンツ取得プロトコルに伝播させ，合成プロトコルを全体を終了させる必要がある．具体

© 2012 Information Processing Society of Japan

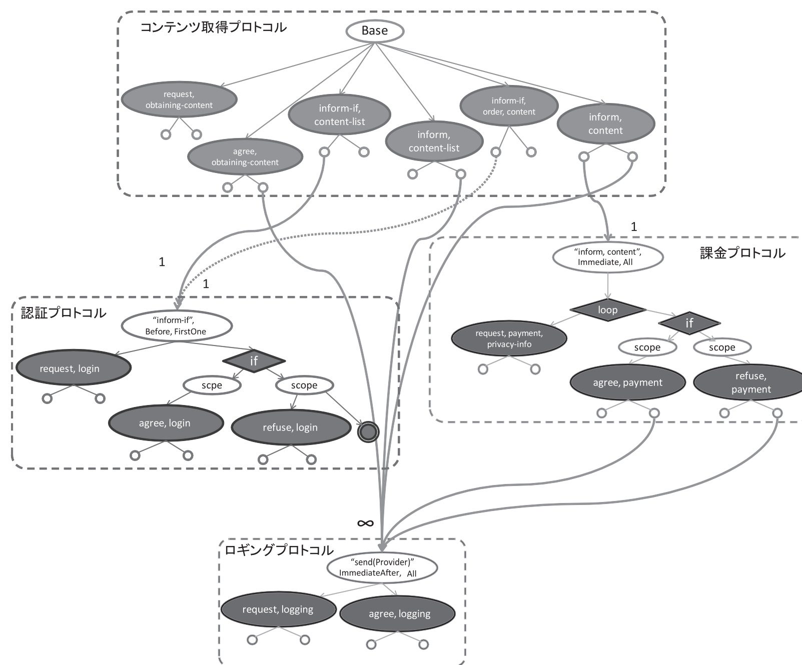

図 9　循環検証ツリー（コンテンツ取得＋認証＋課金＋ロギング）

Fig. 9　Detection tree (Obtaining content + Authentication + Payment + Logging).

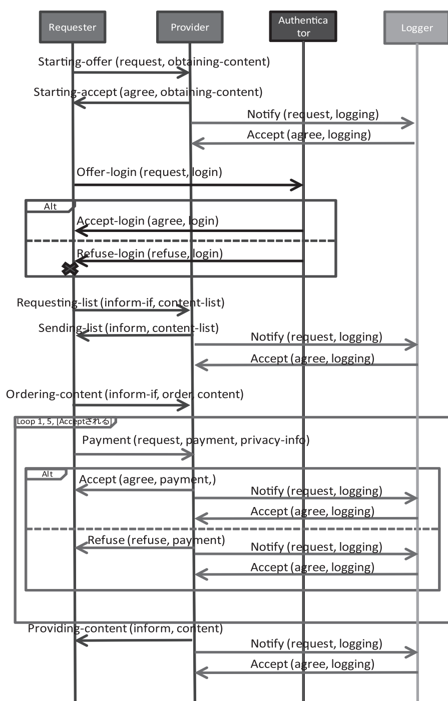

図 10　合成プロトコル（コンテンツ取得＋認証＋課金＋ロギング）

Fig. 10　Compositon protocol (Obtaining content + Authentication + Payment + Logging).

的には，コンテンツ取得プロトコルが本来持っているロールバック例外に伝播させ，コンテンツ取得取り消しを実行したうえで，協調動作全体を終了する．課金プロトコルとコンテンツ取得プロトコルそれぞれの例外記述と，その例外伝播を実現する合成結果を**図 11** に示す．伝播指定では，課金プロトコル（PaymentProtocol）の CancelPayment 例

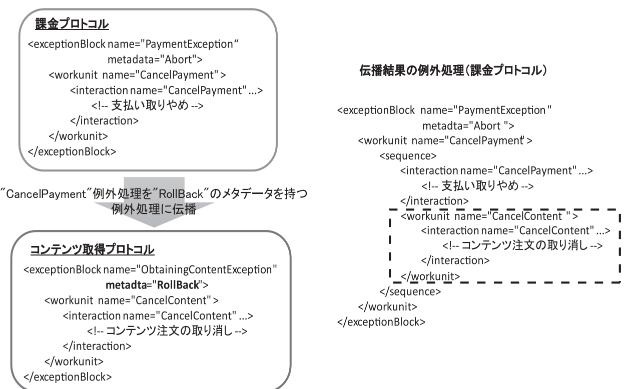

図 11　課金プロトコルからコンテンツ取得プロトコルへの例外伝播

Fig. 11　Propagating exception from payment to obtaining content.

外処理から，その親プロトコルで，“RollBack” メタデータを持つ例外処理へ伝播を行う．これにより，コンテンツ取得プロトコルの CancelContent 例外処理内で，「支払い取りやめ」→「コンテンツ注文の取り消し」という順番で処理が実行され，例外の伝播が実行される．

また，図 6 に示した，認証処理の失敗時例外処理から親プロトコルの終了処理への伝播も，同様に伝播指定によって実現することができる．

7. 考察

本手法は，メタデータを用いて表現した制約を順守する合成方法を探索する自動合成手法である．協調プロトコル合成では，合成プロトコルは合成順序と範囲候補範囲から選択した合成箇所に依存するため，単純な組合せ数は膨大

© 2012 Information Processing Society of Japan

になる．また，最後まで合成を行わないと合成の成否判断ができないため，手動での合成試行はコストが高く，機械的に探索する方法が有効である．ほかにも，従来手法では合成モデルを挟まずWS-CDL上で，直接，合成を行うので，合成箇所の競合を回避することができない．合成箇所の競合は合成順序の試行では回避することができない問題なので，本手法のように複数の部品プロトコルを並列化して合成することが必要である．

本手法はメインとなる協調プロトコルに対して，様々な汎用部品を後から挿入することができる．協調動作設計において，設計者はメインロジックの設計のみに集中することができる．また，ビジネスプロセスにおいて，運用環境の変更等にともなう要求の変更はしばしば発生する．本手法は部品プロトコルが実装済みかそうでないかを問わず，合成に使用でき，部品の付け替えをすることにより，様々な要求を満たす協調プロトコルを作成できるため，要求の変更に柔軟に対応できる．

以下では，本手法での合成および検証の網羅性と例外伝播の有効性について考察する．

7.1 合成の網羅性

メタデータを使用した協調プロトコル合成は，合成の適用順序に大きく依存している．制約の表現において，*FirstOne* や *LastOne* といった実行順序に関わる制約は，最終的な合成プロトコルのメッセージ列を表している．したがって，合成時の段階で制約を満たしていたとしても，最終的な合成プロトコルでは制約を満たせていない可能性がある．これは，合成を経ることで協調プロトコルのメッセージが増えていくことと，各合成許可制約の適用時は，その段階でのメッセージ構成を基に合成を行っているためである．

そこで，提案手法では各部品プロトコルの合成段階では，網羅的に合成を試み，最後に制約を検証する手法をとっている．各合成許可制約の適用順や，合成箇所範囲を網羅的に探索している．具体的には各合成時に制約が満たされていたのが，後の合成で崩された場合，競合しないような別の合成箇所を探索したり，合成順序を変更したりして，制約を崩す合成を先に実行するような適用順序での合成をも試みる．

ただし，制御構造による合成候補範囲の修正によって，不必要な合成候補を削除し，効率化も行われている．制御構造共通の修正ルールである上位スコープへの合成は行わないというのは，基点よりも上位のスコープに挿入を行うと，いずれかのパスにおいて，基点メッセージを通ることなく，合成箇所だけを通るパスが発生してしまうからである．1つでも基点メッセージを通ることなく，合成部分が実行されてしまうパスがあれば，それは制約違反である．基点メッセージより下位のスコープを形成するループ構造への合成を行わないのも同様の理由からである．ループ構造の場合，ループ回数が0回のときに，合成が行われていても合成箇所が実行されなくなるため，制約の順守を保証できなくなっている．したがって，これらすべての実行パスで制約が保証できなくなる合成範囲には，あらかじめ合成を行わなくし，試行回数の軽減を行っている．

このように，合成順序に依存度の高いメタデータを使用した協調プロトコル合成において，本手法は網羅的に合成を試み，検証によって制約が順守されていることを保証している．

7.2 検証の網羅性

検証は，作成した合成プロトコルの構造をパス網羅によって展開することで，実行される可能性のある実行パスをすべて取得し検証する．パス網羅で分岐パスの組合せを網羅的に確認しているので，検証をパスすれば，どのような実行パスでも制約を満たすことになる．この検証手法では，制約表現で指定している合成回数を，部品プロトコルの合成回数を実行回数で確認している．しかし，協調プロトコル上では，実行時と同じ条件判断をすることができない．したがって，分岐構造の各分岐パスは，等しく実行される可能性があると見なし，すべてのパスを検証している．分岐条件によって，絶対に実行されない分岐パスがあったとしても，そこはすべてのパスが実行されうるものとし，かつ制約を満たしていることが必須となる．ループ構造も分岐構造と同様にループ条件を検証し，ループ回数を判断し検証することはできない．そこで，ループ部分が実行された場合とされなかった場合の2通りの場合を検証し，そのいずれでも制約が満たされていることを確認する．合成許可制約で記述するのは，あくまで合成回数もしくは合成箇所数であるので，ループによって同じ処理が何回実行されるかというのは，制約には関係なく，ループ部分は最大でも1回のみ実行すればよい．逆にいえば，本手法においては，必ず1回のみ実行されるというような合成は指定できない．

協調プロトコルの検証方法としては，既存のモデル検査ツールを使用する方法も考えられる．Zhaoら [7] の研究では，WS-CDLで記述されたをPromelaのモデルに変換する手法を提案している．Promelaモデルは，モデル検査ツールであるSPIN [8] を用いることにより，自相論理の検証が可能となる．本手法で定義した合成許可制約や合成禁止制約を，自走論理に変換することで，SPIN上での検証が可能になると考えられる．

7.3 例外伝播の有用性

例外伝播は部品プロトコルのモジュール性を維持しつつ，再利用性を高めるのに重要な意味を持つ．例外伝播によって，部品プロトコル自体は自身の責務となる範囲でし

© 2012 Information Processing Society of Japan

か例外処理や終了処理を持つ必要がなく，それらが集まることでシステム全体の例外・終了処理を構成することができる．

本手法で定義した例外伝播では，親プロトコルの正常系処理，例外処理，終了処理の 3 つを伝播先としている．この分岐先は，終了するか継続するかの 2 種類に大別できる．終了においても正常終了か異常終了かの区別はシステムの運用・保守等の観点から大きな意味を持つ．親プロトコルに対して，部品プロトコルの例外処理や終了処理がどのような影響を持つか考慮し，継続可能か終了かを選択できることは堅牢なシステムの実現につながる．

また，伝播実行後の継続に関しては，本手法では処理戻り先が，親プロトコルが子プロトコルを呼び出した箇所に限定される．ここでさらなる継続方法の要求を考えると，たとえば認証処理の場合等は，認証が失敗して例外処理が実行されたら，ある時点まで戻して再試行させたいという要求も考えられる．このような再試行の伝播等を考えると，正常系処理の任意の箇所に処理を飛ばすという伝播方法が考えられる．しかし，この伝播方法の実現には WS-CDL の拡張をしなければならないうえ，伝播指示に際して部品プロトコルのさらなる理解も必要となる．任意箇所に処理を飛ばすことによってモジュール性が低下し，部品プロトコルの性質を損なう場合が発生しうる．そのために，合成者が部品プロトコルの詳細を理解し，部品プロトコルの性質が損なわれないという確信のもとに伝播させることが必要となるが，これは本研究の合成者の協調プロトコル構築コストの低減という目標に反するため，本研究では対象外とする．

8. 関連研究

Web サービスを複数組み合わせて，新たな Web サービスやビジネスロジックを構築する研究は様々なものがある．Web サービス等の複数の主体の協調動作設計には大きく分けて 2 種類の表現方法がある．Orchestration 型と Choreography 型である．Orchestration 型はリーダとなる 1 つのサービスが，ユーザやビジネスロジックを構成する各サービスとのやりとりを一手に引き受ける形で，複数の主体の連携を実現する．データの受け渡し等はリーダを経由して渡されるので，そこがボトルネックになる可能性があるが，各サービスはリーダの存在のみを知っていれば，あらゆる連携動作に参加することができる．BPEL [9]，BPMN [10] の一部等がこの形式で，協調動作を実現している．もう一方の Choreography 型ではリーダとなるサービスはない．実行結果のデータは，次のサービスへ直接渡される．各サービスが，協調相手や手順を理解していないと協調動作を実現することはできないが，ボトルネックはない．WS-CDL [5] や BPEL4Chor [11]，OWL-P [1] 等がこの形式で協調動作を定義・実現している．本手法では，WS-CDL を協調プロトコルの記述言語に選択した．これは，複数の企業等の主体が連携することを想定しているため，リーダを必要としない Choreography 型の方が，適切な相手とのみデータの授受が行われ，より責務を明確にした協調動作設計ができると考えたためである．また，特定の実装言語に依存しないという WS-CDL の特徴により，部品プロトコルの高い再利用性も見込める．

本手法は複数の小規模・単機能な協調動作を組み合わせることで，複雑な協調動作の設計を目的としている．ボトムアップの合成による協調プロトコル設計手法である．それに対して，Mazouzi ら [3] はトップダウンの設計手法を提案している．協調プロトコル中の抽象処理と定義してあった箇所に，後から具体的な処理方法を含む部品を後から組み合わせることで，状況に応じて処理方法を切り替えることができる手法を提案している．トップダウンの手法では，実行環境等に合わせて，処理の具体的な方法を柔軟に変更することができるが，その合成箇所はあらかじめ抽象処理として定義しておく必要がある．本手法は，部品プロトコルの段階では，具体的な合成方法等を考える必要はない．したがって，部品プロトコルの高い再利用性で，柔軟が合成を行うことができる．

複数の分散された主体が協調動作するビジネスプロセスにおいて，信頼性の確保は重要なことである．メインロジックだけでなく，提案手法で合成部品の対象としている，ロギングや認証といった処理は信頼性の高いビジネスプロセスを設計するうえでは非常に重要なことである．Yao ら [12] の手法では，本手法で用いたように，ロギングを各サービスの実行前後に挟み込むことによって，構成サービスの監視やポリシ違反時の対処処理の実行を行うことができる．Yao らの手法は，あくまでサービス品質の監視および品質回復にのみ焦点を当てているため，本手法のように，コンテンツ取得サービスに課金処理を合成することで，コンテンツ購入サービスに変更するといったメインロジック自体の変更を行うことはできない．

提案手法は，協調動作のメインロジックを定義している協調プロトコルに対して，ロギングや課金といった汎用的な処理を後から挿入指定し，組み合わせる手法を提案している．プログラミング手法の 1 つである，AOP（Aspect Oriented Programming）ライクな合成を協調プロトコル上で実現しているといえる．提案手法と同様，AOP ライクな協調プロトコル合成の既存研究がいくつかある．BPEL’n’Aspects [13] は BPEL 上で AOP ライクな協調プロトコルの合成を行う手法である．提案手法同様，協調プロトコルの合成を行う手法であるが，合成箇所を指し示すポイントカットの指定に，BPEL の記述要素であるイベント名を直接記述する方式をとっている．また，Niemöller ら [14] の研究も同様に，AOP ライクな手法を用いて，協調プロトコルの合成を行っている．Niemöller らの手法は，

© 2012 Information Processing Society of Japan

ポイントカットに，“ビジネスプロセスを構成する各タスクにサービスが割り当てられた後” や，“サービスを実行した結果の返り値を受信したら”，といったサービスが共通して持つタイミングをポイントカットとして指定することができる．これらの手法は，実行可能なビジネスプロセスモデリング言語である BPEL を AOP ライクに変更することで，実行時の動的な振舞い変更を実現している．対して，提案手法は設計段階における協調動作設計を対象としているため，ビジネスプロセス定義後に各サービスの実装が必要になる．その分，メタデータを用いることで既存手法よりもアプリケーションよりの意味や特徴をとらえた合成ができる．

提案手法の特徴の１つとして，合成制約を書き換えることによって，様々なバリエーションの協調プロトコルを容易に作成することができる．ビジネスプロセスにおいて，要求や状況が変化することはよくあり，それらの変化に柔軟に対応できるのは大きなメリットである．ビジネスプロセスの要求変化に応じて，協調プロトコルを柔軟に変更する研究には，Mahfouz ら [15] の研究がある．Choreography で表現された協調動作を，要求変化に応じでどのように修正するか，また，グローバルな視点である Choreography の変更に対して，ローカルな視点である各主体のプロセスを変更させる方法を提案している．提案手法は部品プロトコルという形で，処理があらかじめコンポーネント化されていて再利用できることを前提として，合成者のコスト低減を目的としている．したがって，部品プロトコルとして再利用しにくい処理を追加したい場合や，環境に合わせた細粒度の修正を行う場合は，Mahfouz らのような手動によるビジネスプロセスの修正手法が必要となる．

9. まとめ

本論文では，協調プロトコルの自動合成手法を提案した．自動合成では，合成者は使用する部品プロトコルと合成制約を合成器に与えるだけでよく，制約を満たす合成方法を合成器が自動で探索し，合成を行う．合成制約はメタデータを用いて表現されている．メタデータは協調プロトコルを構成するメッセージの意味や性質を表現しており，メタデータを使用することで，より制約を抽象的に表現することができる．合成は，合成順序や合成箇所の探索を網羅的に行い，最終的にできあがった合成プロトコルに対する制約検証によって合成者の望む性質の保持を保証している．また，本手法では正常系処理の合成だけでなく，例外処理や終了処理の合成も実現している．これら例外処理や終了処理の合成も，メタデータを用いて指定・判断しており，柔軟な合成の仕組みを提案している．本手法を使用することで，合意者は作成したい合成プロトコルの性質をメタデータを用いて表現することで，容易に目的の協調プロトコルを得ることができる．

現在扱っている協調プロトコルは，実装をともなわないものや，実装をともなっていても合成によって実装してある処理手順を変更しないよう保護したものを対象にしている．将来課題として，実装部分にまで合成対象を広げ，合成後すぐに実行可能な協調動作設計を作り出すことが考えられる．これにより，実行時の状況に合わせた，柔軟な処理の切替えが見込める．

参考文献

[1] Desai, N. et al.: OWL-P: A Methodology for Business Process Development, *AOIS*, pp.79–94 (2005).

[2] Vitteau, B. and Huget, M.-P.: Modularity in Interaction Protocols, *Workshop on Agent Communication Languages*, pp.291–309 (2003).

[3] Mazouzi, H. et al.: Open protocol design for complex interactions in multi-agent systems, *Proc. 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, pp.517–526 (2002).

[4] 高橋竜一，鄭　顕志，石川冬樹，本位田真一，深澤良彰：マルチエージェントシステムにおけるメタデータを用いた協調プロトコル合成手法，電子情報通信学会論文誌 D，ソフトウェアエージェントとその応用論文特集，Vol.J92-D, No.11, pp.1827–1839 (2009).

[5] Kavantzas, N., Burdett, D., et al.: Web Services Choreography Description Language Version 1.0., available from ⟨http://www.w3.org/2002/ws/chor/edcopies/cdl/cdl.html⟩.

[6] FIPA: FIPA ACL Message Structure Specification, available from ⟨http://www.fipa.org/specs/fipa00061/SC00061G.html⟩.

[7] Zhao, X. et al.: Verification of WS-CDL Choreography, *Proc. 1st Asian Working Conference on Verified Software*, pp.163–178 (2006).

[8] Holzmann, G.J.: *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional (2003).

[9] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I. and Weerawarana, S.: *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*, IBM (2003).

[10] OMG: Business Process Model and Notation (BPMN) Version 2.0, available from ⟨http://www.bpmn.org/⟩ (2011).

[11] Decker, G. et al.: BPEL4Chor: Extending BPEL for Modeling Choreographies, *The IEEE International Conference on Web Services*, pp.296–303 (2007).

[12] Yao, J., Chen, S., Wang, C., Levy, D. and Zic, J.: Accountability as a Service for the Cloud, *IEEE International Conference on Services Computing*, pp.81–88 (online), DOI: http://doi.ieeecomputersociety.org/10.1109/SCC.2010.83 (2010).

[13] Karastoyanova, D. and Leymann, F.: BPEL'n'Aspects: Adapting Service Orchestration Logic, *Proc. 2009 IEEE International Conference on Web Services, ICWS'09*, pp.222–229, IEEE Computer Society, Washington, DC, USA (online), DOI: http://dx.doi.org/10.1109/ICWS.2009.75 (2009).

[14] Niemöller, J., Levenshteyn, R., Freiter, E., Vandikas, K., Quinet, R. and Fikouras, I.: Aspect Orientation for Composite Services in the Telecommunication Domain, *Proc. 7th International Joint Conference on Service-Oriented Computing, ICSOC-ServiceWave'09*, pp.19–

© 2012 Information Processing Society of Japan

33, Springer-Verlag, Berlin, Heidelberg (online), DOI: http://dx.doi.org/10.1007/978-3-642-10383-4_2 (2009).
[15] Mahfouz, A., Barroca, L., Laney, R. and Nuseibeh, B.: Requirements-Driven Collaborative Choreography Customization, *Proc. 7th International Joint Conference on Service-Oriented Computing, ICSOC-ServiceWave'09*, pp.144–158, Springer-Verlag, Berlin, Heidelberg (online), DOI: http://dx.doi.org/10.1007/978-3-642-10383-4_10 (2009).

髙橋 竜一

2007年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．2008年同大学大学院基幹理工学科修士課程修了．2011年同大学院基幹理工学研究科後期博士課程単位取得退学．2011年より早稲田大学メディアネットワークセンター助教．協調プロトコル設計，サービス工学の研究に従事．

石川 冬樹 （正会員）

2002年東京大学理学部情報科学科卒業．2007年同大学大学院情報理工学研究科博士課程修了．2007年より国立情報学研究所コンテンツ科学研究系助教，2011年より同准教授，現在に至る．2007年より総合研究大学院大学複合科学研究科助教兼任，現在に至る．サービスコンピューティング，ソフトウェア工学の研究に従事．

本位田 真一 （正会員）

1978年早稲田大学大学院理工学研究科修士課程修了．（株）東芝を経て2000年より国立情報学研究所教授，2004年より同研究所アーキテクチャ科学研究系研究主幹を併任，現在に至る．2001年より東京大学大学院情報理工学系研究科教授を兼任，現在に至る．

深澤 良彰 （正会員）

1976年早稲田大学理工学部電気工学科卒業．1983年同大学大学院博士課程修了．1987年早稲田大学理工学部助教授．1992年同教授．工学博士．ソフトウェア再利用技術を中心としてソフトウェア工学の研究に従事．

© 2012 Information Processing Society of Japan