情報処理学会論文誌 Vol.54 No.2 774–786 (Feb. 2013)

参照整合性の検証のためのUML Activity解析方法

井上 拓[1,a)]　本位田 真一[2,3]

受付日 2012年5月14日, 採録日 2012年11月2日

概要：情報システムのコンポーネントベース開発において，データコンポーネント間にまたがる参照整合性が定義され，UML の *Activity* 図で記述されたサービスコンポーネントの振舞いによって実現される．参照整合性を保つために *Activity* の検証が必要であるが，多数の *Activity* の組合せを人手で網羅的に確認することは困難である．本論文では，*Activity* の静的解析による，参照整合性の自動検証手法を提案する．この手法は，Field-sensitive なデータフロー解析を行い，データアクセスを行う *Action* 記述の順序を解析する点が特徴である．本手法を用いて，仕様定義段階で *Activity* の欠陥を検出することが可能になる．これにより，データの一貫性が保たれた安全な情報システムを構築するための足がかりが得られる．

キーワード：コンポーネント，参照整合性，UML Activity，データフロー解析

UML Activity Analysis for Verifying Referential Integrity

Taku Inoue[1,a)]　Shinichi Honiden[2,3]

Received: May 14, 2012, Accepted: November 2, 2012

Abstract: In component-based IS (information systems), the referential integrities across data-components are realized by the behaviors of service-components described in UML activities, and it is a hard task to verify numerous combinations of activities by hand. This paper presents a static analysis method of UML Activities for automated verification of referential integrities, which employs a field-sensitive data-flow analysis and a sequence analysis of actions accessing data. With the method we can detect the defects in activities, which gives us a foothold to construct the consistent IS.

Keywords: component, referential integrity, UML Activity, data-flow analysis

1. はじめに

情報システムの開発において，Catalysis [1]，UML Components [2] 等の，Unified Modeling Language（UML）を用いたコンポーネントベース開発方法論が数多く適用されている．コンポーネントベース開発において関心の分離は重要な概念であり，これらの開発方法論では，情報システムの主要な関心事であるデータの管理とビジネスゴールの実現を，データコンポーネント（DC）とサービスコンポーネント（SC）の 2 種類のソフトウェア部品に集約して，システムを構築する．DC はシステムが扱うデータを永続化・管理する責務を持ち，データにアクセスするための操作を提供する．一方 SC は，DC のデータにアクセスし，ビジネスゴールを実現するためのシステム機能をサービスとして提供する．サービスの処理フローの仕様は，UML の *Activity* 図で記述される．

一般的な情報システムには，データオブジェクトの参照整合性 [3] や，ライフサイクルの整合性 [4] 等の，データの整合性が定義される．参照整合性はデータの一貫性を保証する重要な制約であり，ときに複数の DC にまたがって定義される．多くの場合，DC の再利用性を保つために，個々の DC は自身が管理するデータの整合性のみ保証する．DC にまたがる参照整合性は，SC の *Activity* の中で，参照元と参照先の DC のデータアクセス操作を呼び出す *Action*

1 キヤノン株式会社
Canon Inc., Ota, Tokyo 146–8501, Japan
2 東京大学
The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
3 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
a) taku.inoue@gmail.com

© 2013 Information Processing Society of Japan

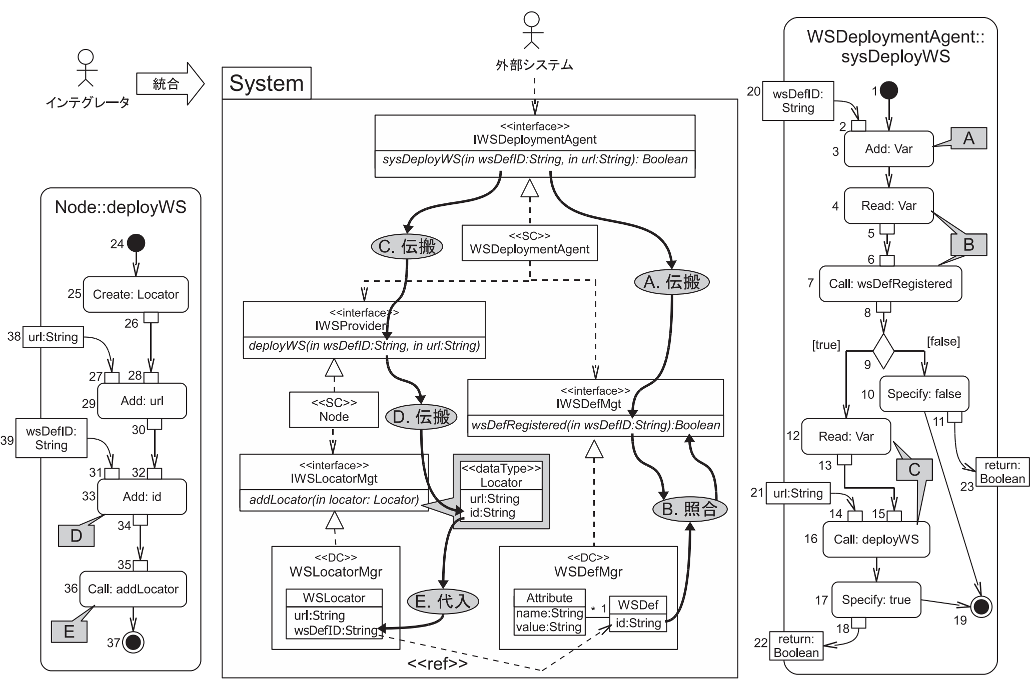


図 1　2 種類のコンポーネントとデータの参照関係

Fig. 1　Two types of component and data reference.

を適切な順序で組み合わせることによって実現される．

図 1 の例を用いて説明する．図 1 中の ⟨⟨ref⟩⟩ は，2 つの DC（WSLocatorMgr，WSDefMgr）が管理するデータの属性（WSLocator.wsDefID，WSDef.id）の間の参照整合性[*1]を示しており，2 つの SC（WSDeploymentAgent, Node）の操作 sysDeployWS，deployWS の *Activity* の *Action* 記述によって実現される[*2]．まず sysDeployWS は，操作パラメータ wsDefID の値を変数 Var に書き込み（手順 A），Var の値を引数 wsDefID として DC の操作 wsDefRegistered を呼び出し（手順 B），wsDefID の値を属性 id に持つ WSDef オブジェクトの存在を照合する．そして Var の値を引数 wsDefID に指定して，操作 deployWS を呼び出す（手順 C）．操作 deployWS では，パラメータ wsDefID の値を Locator オブジェクトの属性 id に書き込み（手順 D），該オブジェクトを引数 locator として DC の操作 addLocator を呼び出し（手順 E），id の値を WSLocator オブジェクトの属性 wsDefID に代入する．この例では，手順 E の代入の *Action* に先立って，手順 B で照合の *Action* を実行することによって，代入の *Action* が，参照整合性を破壊しない安全な文脈で実行されることを保証している．

UML を用いたモデルベースの開発では，仕様定義段階でデータの整合性を確保することが重要であり，部品を統合するインテグレータは，SC の *Activity* の組合せが参照整合性を満たすことの確認を行う．しかし *Activity* の詳細な知識は SC の開発者に分散しており，さらに *Activity* の組合せは各システム機能に対応して数多く存在する．人手で誤りなく網羅的に整合性を確認することは困難であり，自動化された検証手法が必要である．

Planas らは文献 [5], [6] において，*Action* 記述の列 ⟨n,act⟩+ のデータ整合性の自動検証手法を提案している．n はアクション act の実行回数であり，+は繰返しを表す．この手法は，データ整合性を破壊する可能性がある *Action* の種別ごとに，整合性を満たすために必要な補完 *Action* を定義して，列上の各要素の補完 *Action* が列に包含される場合に，整合性が保たれると判定する．各 *Action* が作用する対象データは，*Action* の引数に与えるインスタンスの参照から特定する．たとえば必須属性 att を持つクラス cls のインスタンス o の生成を表す o:=CreateObject(cls) の補完 *Action* は，o.att に値 v を設定する AddStructuralFeature(o,att,v) である．

Planas らの手法は，単一のデータモデルの整合性の検証に有用であるが，複数の DC 間のデータの整合性と，*Action* の順序関係を考慮しておらず，DC 間にまたがる参照整合性の検証には適用できない．また前述の列の形式は，複数の *Action* 記述を含むループ等の，一般的な *Activity* の制御構造を表現する能力を持たない．そこで本論文では，補完 *Action* のアイデアを複数の DC 間に拡張して，*Activity* を対象とする，DC 間にまたがる参照整合性の自動検証手法を提案する．

DC 間にまたがる参照整合性を扱うために，整合性を破壊しうる破壊的なデータアクセスと，破壊を防ぐ補完的アクセスの組合せの定義が必要である（技術課題 1）．本手法

[*1] WSLocator インスタンスの wsDefID と同じ値を id に持つ WSDef インスタンスが存在する，という整合性．

[*2] 説明の都合上，図に実行順を示すアルファベット記号（A–E）と補助線を付加してある．*Activity* の各ノード要素の識別番号（1–39）は，2 章以降の説明において参照する．

© 2013 Information Processing Society of Japan

では，関係データベースの参照整合性と，その実現方法の類比からアクセスの定義を行い，この課題を解決する．また我々の問題では，*Activity* で DC 内のインスタンスの参照を直接扱わないため，図 1 のようなオブジェクトの属性値の伝搬をたどって，アクセス操作に与える引数の由来を特定する必要がある（技術課題 2）．本手法では，*Activity* の Field-sensitive なデータフロー解析の方法を導入して，この課題を解決する．さらに，破壊的アクセスと補完的アクセスの順序関係を考慮した，*Activity* の検証方法が必要である（技術課題 3）．本手法では，*Actvity* の制御フローに沿って，アクセス操作の呼び出し履歴を静的に解析する方法を導入して，この課題を解決する．

本論文では，上記 3 つの課題に対する提案手法の妥当性と検証能力を議論し，手法の有効な利用方法を述べる．さらに実問題の仕様モデルに本手法を適用し，有用性を評価する．なお本論文では，システムが提供する機能ごとの整合性を検証の対象とし，複数機能を同時実行する場合の並行性に関する検証は行わない．

本論文の構成は以下のとおりである．2 章で本手法が扱うコンポーネントの仕様モデルを説明する．3 章で提案手法を説明し，4 章で手法の妥当性と検証能力，有用性を議論する．5 章で関連研究について述べ，6 章で本論文をまとめる．

2. 仕様モデル

本章では，提案手法が扱う SC，DC の仕様モデルを定義する．仕様モデルは，静的な側面を記述した構造モデルと，操作の振舞いモデルからなる．DC に関しては，振舞いモデルから導出されるアクセス情報が与えられる．

構造モデル

データモデルと，コンポーネントが提供・使用するインタフェースをクラス図で表現する．永続データの管理を行わない SC には，インタフェース記述のみ定義される．

データモデルは，DC が管理するデータを *Class* として，データの関係を関連として表現する．各 *Class* は属性（*Property*）の定義を持つが，操作を持たない．システム内の DC 間にまたがる参照は関係 $REF \subseteq Class \times Property \times Class \times Property$ で与えられる．$r = (cls_{src}, prop_{src}, cls_{dst}, prop_{dst}) \in REF$ は，属性 $cls_{src}.prop_{src}$ から $cls_{dst}.prop_{dst}$ への参照を表す．ここで $prop_{dst}$ は cls_{dst} のインスタンスを識別するキー属性であり，インスタンスのライフサイクルを通じて値が不変であるものとする．図 1 の例題では $REF = \{(\mathrm{WSLocator}, \mathrm{wsDefID}, \mathrm{WSDef}, \mathrm{id})\}$ である．本論文では以降の章で，例題における唯一の参照を r_0 として説明を行う．

インタフェースは UML の *Interface* として表現され，SC，DC が提供する操作のシグニチャとパラメータの型が定義される．本手法では，各コンポーネントが異なるノードに分散配置され，操作パラメータは値渡しされることを想定する．したがって，操作パラメータの型は *DataType*，*PrimitiveType*，*Enumeration* のいずれか，またはその *Collection* である．in/out はパラメータの方向を示す．SC は DC の操作の呼び出しを通じて，DC のデータにアクセスすることができるが，データは値渡しの操作パラメータにシリアライズされ，DC 内のオブジェクトを直接参照することはできない．これによりコンポーネント間の結合が疎に保たれる．

SC の振舞いモデル

SC の振舞いモデルは，SC の操作の処理フローを UML の *Activity* として表現する．*Activity* はノードとエッジで構成されるグラフである．本手法が扱う *Activity* の要素を**図 2** に示す．ノードは，制御点を表す *ControlNode*，オブジェクトを表す *ObjectNode*，処理を表す *ExecutableNode* に分類される．*Action* は単一の処理を表し，*StructuredActivityNode* は複数のノードを組み合わせた処理を表す．*Variable* は一時的な情報を保持するローカル変数である．

Action の分類 1–5 は各々，オブジェクトの生成・消滅，属性値の追加・削除・取得，変数値の追加・削除・取得，値の指定，操作の呼び出しを表す．分類 4 の *ValueSpecificationAction* は，図 1 のノード 10 の false，ノード 17 の true のような，オブジェクトやデータの具体的な値を指定するために用いられる．分類 5 の *CallOperationAction* の操作の呼び出し方式は，SC で広く採用されている同期方式に限定する．すなわち，呼び出し側は操作の完了まで待機する．

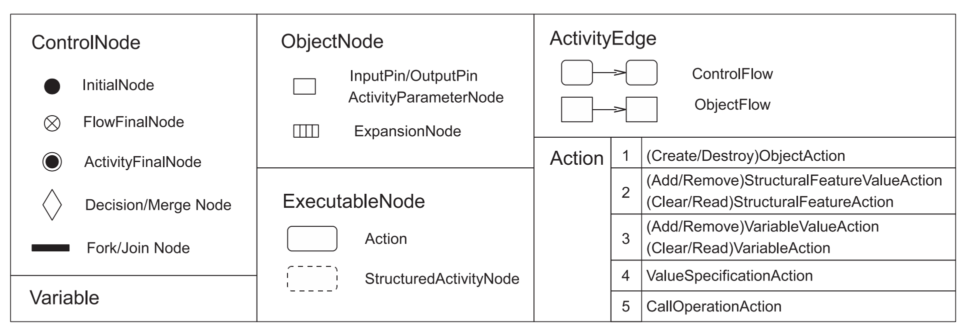


図 2 *Activity* の要素
Fig. 2 *Activity* elements.

© 2013 Information Processing Society of Japan

```
1  context WSLocatorMgr::addLocator(in locator:Locator)
2  post: WSLocator.allInstances()->one(i|i.url=locator.url and i.wsDefID=locator.id)
3  context WSDefMgr::wsDefRegistered(in wsDefID:String):Boolean
4  post: result=WSDef.allInstances()->exists(i|i.id@pre=wsDefID)
```

図 3 DC の操作の OCL 記述
Fig. 3 An OCL description of data-component operations.

DC の振舞いモデルとアクセス情報

DC の振舞いモデルは，DC の操作を介したデータへのアクセスの仕様を表現する．本論文では，開発方法論 [1], [2] や文献 [7] と同様に，DC の振舞いモデルが，宣言的な仕様記述言語である Object Constraint Language（OCL）を用いて，操作の事前条件・事後条件の形式で記述されることを想定する．図 1 の例題における DC の操作 wsDefRegistered, addLocator の振舞いモデルを**図 3** に示す．

本手法は，DC の振舞いモデルから導出される，データへのアクセス情報 $AC = (A, G, D, C)$ を利用する．A, G, $D \subseteq$ $Operation \times Class \times Property \times Parameter \times (Property \cup$ NULL)，$C \subseteq Operation \times Class \times Property \times Parameter \times$ $(Property \cup$ NULL$) \times Parameter \times (Property \cup$ NULL$)$ である．

- A を代入情報と呼ぶ．$a = (op, cls, prop_{cls}, param, prop_{param}) \in A$ は，操作 op の実行を通じて，入力パラメータの属性 $param.prop_{param}$ の値が，クラス cls のインスタンスの属性 $prop_{cls}$ に代入されることを表す．
- G を取得情報と呼ぶ．$g = (op, cls, prop_{cls}, param, prop_{param}) \in G$ は，操作 op の実行を通じて，クラス cls のインスタンスの属性 $prop_{cls}$ の値が，出力パラメータの属性 $param.prop_{param}$ として得られることを表す．
- D を削除情報と呼ぶ．$d = (op, cls, prop_{cls}, param, prop_{param}) \in D$ は，操作 op の実行を通じて，入力パラメータの属性値 $param.prop_{param}$ を属性 $prop_{cls}$ に持つような，クラス cls のインスタンスが，削除されることを表す．
- C を照合情報と呼ぶ．$c = (op, cls, prop_{cls}, param_{in}, prop_{in}, param_{out}, prop_{out}) \in C$ は，操作 op の実行を通じて，入力パラメータの属性 $param_{in}.prop_{in}$ の値を属性 $prop_{cls}$ に持つような，クラス cls のインスタンスの存在を照合し，照合結果が出力パラメータの属性 $param_{out}.prop_{out}$ の値として得られることを表す．

パラメータ自身の値がデータのアクセスに用いられる場合は，パラメータの属性の代わりに NULL が指定される．

AC は，OCL 記述の意味の解釈に基づいて導出される．図 3 の OCL 記述例を用いて説明する．図 3 の 1, 2 行目は，操作 addLocator の実行を通じて，入力パラメータ locator の属性 url（id）の値が，WSLocator のあるインスタンス i の属性 url（wsDefID）に代入されることを表す．したがって，$A = \{$(addLocator, WSLocator, url, locator, url), (addLocator, WSLocator, wsDefID, locator, id)$\}$ が導出される．また 3，4 行目は，操作 wsDefRegistered の実行を通じて，入力パラメータ wsDefID の値を属性 id に持つ WSDef インスタンスの存在を照合し，その結果が戻り値 result として得られることを示しており，$C = \{$(wsDefRegistered, WSDef, id, wsDefID, NULL, return, NULL)$\}$ が導出される．データの削除と取得のアクセスは図 3 に記述されていないため $G = D = \phi$ である．

DC の各操作から，0 個以上の複数の A，G，D，C の要素が導出される．AC の導出は，DC の振舞いを定義する仕様記述者が人手で実施する必要があるが，A，G に関しては，筆者らが文献 [8] で提案した方法を用いて自動で導出することも可能である．本手法では，特定の導出方法を仮定せず，AC を手法の入力として想定する．

SC の *Activity* の作成者は，OCL 記述の内容に基づいて，DC のデータアクセス操作の呼び出しを，*CallOperationAction* を用いて *Activity* に記述する．この呼び出し記述は，対象操作の OCL 記述から導出される A，G，D，C の要素が示すアクセスの実行を表す．たとえば，図 1 の番号 7 の *CallOperationAction* ノードは，DC の操作 wsDefRegistered の呼び出しを記述し，図 3 の wsDefRegistered の OCL 記述から導出される (wsDefRegistered, WSDef, id, wsDefID, NULL, return, NULL) $\in C$ の照合アクセスの実行を表している．

3. 提案手法

本手法は，構造モデル，SC の振舞いモデル，DC のアクセス情報を入力として，SC の振舞いモデルの，参照の集合 *REF* に対する整合性を検証する．手法のプロセスは，操作呼び出しのコールグラフを生成する Step 1，参照整合性を破壊しうる破壊的アクセスと，破壊を防ぐための補完的アクセスの組合せを定義する Step 2，*Activity* のデータフロー解析を行う Step 3，整合性の検証を行う Step 4 の 4 ステップからなる（**図 4**）．全システム機能に対して Step 1，2 を 1 度だけ実施した後，システム機能ごとに Step 3，4 を繰り返し実施する．

Step 2 で，関係データベースの参照整合性とその実現方法の類比から，アクセス操作の定義を行い，課題 1 を解決する．Step 3 で，Field-sensitive な *Activity* のデータ

© 2013 Information Processing Society of Japan

情報処理学会論文誌　Vol.54　No.2　774–786 (Feb. 2013)

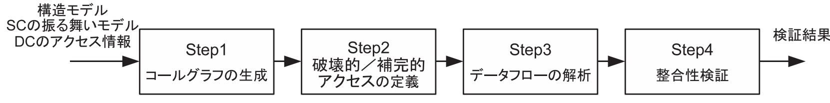


図 4　提案手法のプロセス
Fig. 4　The process of our method.

$$CG = \{(op, op', a')|op, op' \in Operation \wedge a' \in CallOperationAction \wedge a'.activity.specification = op' \wedge a'.operation = op)\}$$

図 5　コールグラフ CG
Fig. 5　Call-graph CG.

フロー解析の方法を導入し，操作呼び出しの引数の到達定義を求めることにより，課題 2 を解決する．Step 4 では，*Actvity* の制御フローをたどって，各ノードにおけるアクセス操作の呼び出し履歴を求め，その結果をもとに整合性の判定を行うことにより，課題 3 を解決する．以下に各ステップの処理内容を述べる．

3.1　Step 1：コールグラフの生成

各システム機能は，複数の SC，DC の操作の振舞いを組み合わせて実現され，一連の操作の実行が完了した時点で，参照整合性が満たされていなければならない．そこで本ステップで，システム機能ごとの SC，DC の操作の呼び出し関係をコールグラフ $CG \subseteq Operation \times Operation \times CallOperationAction$ として生成し，以降のステップで用いる．コールグラフ CG の定義を**図 5** に示す．$call = (op, op', a') \in CG$ は，SC の操作 op' の振舞いを記述する *Activity* 中の $a' \in CallOperationAction$ が，他の SC または DC の操作 op を呼び出すことを表す．システム機能 F の CG を次の手順で静的に生成する．まず空の CG に，F を提供する SC の操作 op' と，その振舞いを表す *Activity* 中で呼び出す操作 op の組を追加する．op が SC の操作である場合は同様の手続を再帰的に繰り返すことにより，CG が得られる．

図 1 の例題では，操作 WSDeploymentAgent::sysDeployWS が提供する唯一のシステム機能について，$CG = \{(\mathrm{wsDefRegistered}, \mathrm{sysDeployWS}, 7), (\mathrm{deployWS}, \mathrm{sysDeployWS}, 16), (\mathrm{addLocator}, \mathrm{deployWS}, 36)\}$ が得られる．ただし 7，16，36 は，各番号に対応するノードを表す．

3.2　Step 2：破壊的アクセス・補完的アクセスの定義

本論文では，AC が表す DC のデータアクセスの中で，DC 間にまたがる参照整合性を破壊しうるアクセスと，破壊を防ぐために必要なアクセスを，それぞれ破壊的アクセス，補完的アクセスと呼ぶ．本ステップでは，これら 2 種類のアクセスの組合せを定義する．関係データベースにおけるテーブル間の参照整合性の類比から，参照 $r = (cls_{src}, prop_{src}, cls_{dst}, prop_{dst}) \in REF$ の整合性は次の 3 通りのデータアクセスの実行パターンにおいて破壊されうる．

(1)：cls_{src} のインスタンスの属性 $prop_{src}$ に値 x を代入する．属性 $prop_{dst}$ の値が x であるような cls_{dst} のインスタンスが存在しない場合に，代入によって r が破壊される．

(2)：属性 $prop_{dst}$ に値 x を持つ cls_{dst} のインスタンスを削除する．属性 $prop_{src}$ の値が x であるような cls_{src} のインスタンスが存在する場合に，削除により r が破壊される．

(3)：cls_{dst} のインスタンスの属性 $prop_{dst}$ の値を x から x' に変更する．属性 $prop_{src}$ の値が x であるような cls_{src} のインスタンスが存在する場合に，$prop_{dst}$ の変更によって r が破壊される．

2 章で述べたように，本手法では (3) の $prop_{dst}$ の値の変更を想定しないため，(1) と (2) のパターンを扱う．

データアクセスの実行パターンは，SC の *Activity* 中に，DC のデータアクセスの実行を含む処理フローとして記述される．(1) と (2) において r の破壊を引き起こすのは，cls_{src} インスタンスの属性 $prop_{src}$ への代入アクセスと，属性 $prop_{dst}$ の値を指定した cls_{dst} インスタンスの削除のアクセスの実行であり，これらのアクセスを破壊的アクセスと呼ぶ．(1)，(2) の破壊的アクセスは，r と AC の情報から，**表 1** の A_r^{src}，D_r^{dst} として特定される．

(1), (2) における r の破壊を防ぐためには，SC の *Activity* 中で，(1) と (2) の破壊的アクセスの実行に先立ち，参照の反対端のデータに対して，次の 2 つのいずれかのアクセスを実行する必要がある．

- (1) と (2) のアクセスが，r を破壊しない安全な文脈で実行されることを保証する．
- (1) と (2) のアクセスの実行による r の破壊を打ち消す効果を与える．

前者は，(1) の場合は以下の (1-1)，(1-2) の実行パターンによって実現され，(2) の場合は (2-1) のパターンによって実現される．後者は，(1) の場合は (1-3) のパターンによって実現され，(2) の場合は (2-2) のパターンによって実現される．

(1-1)：(1) の代入値を，cls_{dst} インスタンスの $prop_{dst}$ の値から取得する．

(1-2)：(1) の代入値と同じ値を $prop_{dst}$ に持つ cls_{dst} インスタンスを照合する．

(1-3)：cls_{dst} インスタンスの $prop_{dst}$ の初期値として，(1)

© 2013 Information Processing Society of Japan

表 1 参照 r の破壊的アクセス・補完的アクセス
Table 1 Destructive/complementary data-access of r.

パターン	破壊的アクセス・補完的アクセス
(1)	$A_r^{src} = \{(op, cls, prop_{cls}, param, prop_{param}) \in A \mid prop_{cls} = r.prop_{src}\}$
(2)	$D_r^{dst} = \{(op, cls, prop_{cls}, param, prop_{param}) \in D \mid prop_{cls} = r.prop_{dst}\}$
(1-1)	$G_r^{dst} = \{(op, cls, prop_{cls}, param, prop_{param}) \in G \mid prop_{cls} = r.prop_{dst}\}$
(1-2)	$C_r^{dst} = \{(op, cls, prop_{cls}, param_{in}, prop_{in}, param_{out}, prop_{out}) \in C \mid prop_{cls} = r.prop_{dst}\}$
(1-3)	$A_r^{dst} = \{(op, cls, prop_{cls}, param, prop_{param}) \in A \mid prop_{cls} = r.prop_{dst}\}$
(2-1)	$C_r^{src} = \{(op, cls, prop_{cls}, param_{in}, prop_{in}, param_{out}, prop_{out}) \in C \mid prop_{cls} = r.prop_{src}\}$
(2-2)	$D_r^{src} = \{(op, cls, prop_{cls}, param, prop_{param}) \in D \mid prop_{cls} = r.prop_{src}\}$

の代入値と同じ値を代入する．

(2-1)：(2) の $prop_{dst}$ と同じ値を $prop_{src}$ に持つ cls_{src} インスタンスを照合する．

(2-2)：(2) の $prop_{dst}$ と同じ値を $prop_{src}$ に持つ cls_{src} インスタンスを削除する．

上記のデータアクセスの実行パターン (1-2) と (2-1), (2-2) は，関係データベースの参照整合性の実現に広く用いられている，参照整合性制約の RESTRICT による照合チェック，CASCADE による連鎖削除の類比である．その他の (1-1) と (1-3) は，参照整合性制約を用いずに，データベース・アプリケーションのコードで参照整合性を実現するケースの類比であり，SC の *Activity* の一連のシーケンスによって，参照整合性を保証する．

(1-1)，(1-2)，(1-3) において r の破壊を防ぐために実行される，cls_{dst} インスタンスの属性 $prop_{dst}$ の値の取得アクセス，属性 $prop_{dst}$ に関する cls_{dst} インスタンスの照合アクセス，cls_{dst} インスタンスの属性 $prop_{dst}$ への代入アクセスを，(1) の破壊的アクセスに対する補完的アクセスと定義する．同様に (2-1)，(2-2) において実行される，属性 $prop_{src}$ に関する cls_{src} インスタンスの照合アクセス，属性 $prop_{src}$ の値を指定した cls_{src} インスタンスの削除アクセスを，(2) の破壊的アクセスに対する補完的アクセスとして定義する．これらの補完的アクセスはそれぞれ，表 1 の G_r^{dst}，C_r^{dst}，A_r^{dst}，C_r^{src}，D_r^{src} として特定される．

破壊的アクセスと補完的アクセスは，各々の $r \in REF$ について定義される．図 1 と図 3 の例題では，参照 r_0 に関して，$(\mathrm{addLocator}, \mathrm{WSLocator}, \mathrm{wsDefID}, \mathrm{locator}, \mathrm{id}) \in A$, $(\mathrm{wsDefRegistered}, \mathrm{WSDef}, \mathrm{id}, \mathrm{wsDefID}, \mathrm{NULL}, \mathrm{return}, \mathrm{NULL}) \in C$ が，それぞれ (1) の破壊的アクセス，(1-2) の補完的アクセスであり，$A_{r_0}^{src} = \{(\mathrm{addLocator}, \mathrm{WSLocator}, \mathrm{wsDefID}, \mathrm{locator}, \mathrm{id})\}$，$C_{r_0}^{dst} = \{(\mathrm{wsDefRegistered}, \mathrm{WSDef}, \mathrm{id}, \mathrm{wsDefID}, \mathrm{NULL}, \mathrm{return}, \mathrm{NULL})\}$，$D_{r_0}^{dst} = G_{r_0}^{dst} = A_{r_0}^{dst} = C_{r_0}^{src} = D_{r_0}^{src} = \phi$ である．

3.3 Step 3：データフローの解析

補完的アクセスの操作呼び出し *act1* が，破壊的アクセスの操作呼び出し *act2* による参照整合性の破壊を防ぐためには，3.2 節で述べた実行パターンの説明から，*act1*，*act2* の引数と戻り値を表す *Pin* [*3]，あるいはそれらの属性の間に，同値関係が成り立たなければならない．たとえば図 1 の例では，操作 wsDefRegistered の呼び出し（手順 B）の *InputPin*（番号 6）と，操作 addLocator の呼び出し（手順 E）の *InputPin*（番号 35）の属性 id の値が等しくなければならない．

操作呼び出しの引数と戻り値の関係は，*Activity* の *ObjectNode* あるいはその属性の間のデータの伝搬によってモデル化される．そこで本ステップでは，Field-sensitive なデータフロー解析を行い，操作呼び出しの引数の到達定義を求める．本ステップで求めた到達定義の情報を用いて，次の Step 4 の中で，操作呼び出しの引数と戻り値の同値関係の判定を行う．データの伝搬は複数の *Activity* にまたがる可能性がある．本ステップでは各 *Activity* 内の *ObjectNode* 間の依存解析を行った後，*Activity* 間にまたがって到達定義の解析を行う．なお本ステップは，Step 1 で求めた各システム機能のコールグラフ *CG* ごとに 1 度，実施されるが，*ObjectNode* 間の依存解析については，解析内容が *CG* に非依存であるため，初回に実施した結果を 2 回目以降で再利用することが可能である．

ObjectNode 間の依存解析

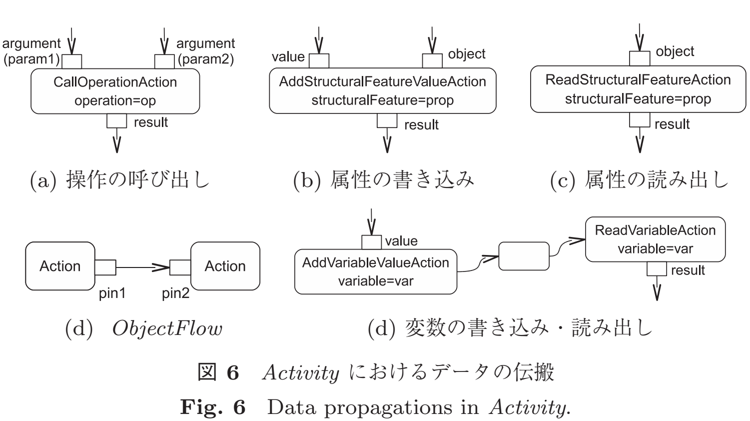
Activity 内の *ObjectNode* 間の依存関係 $IN \subseteq (ObjectNode \times (Property \cup \mathrm{NULL})) \times (ObjectNode \times (Property \cup \mathrm{NULL}))$ を求める．*IN* の定義を**図 7** に示す．$((n, p), (n', p')) \in IN$ は，n' の属性 p' の値が n の属性 p に伝搬することを表す．*IN* の定義の 2 行目の論理式が表すように，n' が属性を持たない場合にのみ $p' = \mathrm{NULL}$ となる．依存関係 *IN* には，次の 3 種類の伝搬が含まれる．

第 1 の伝搬は，**図 6** (d) の例のように *ObjectFlow* によって表現される．*IN* の定義では，n' から n への *ActivityEdge* が存在するという条件（3 行目の論理式）を用いて，この種の伝搬を表している．

第 2 の伝搬は，オブジェクトの属性の書き込み・読み出しによって表現される．図 6 (b) に示す属性の書き込

*3 操作呼び出しは *CallOperationAction* を用いて記述され，引数と戻り値は *InputPin* と *OutputPin* で指定される．各引数は，名前によって操作パラメータと紐付けられる．図 6 (a) において，操作 op を呼び出す *Action* の 2 つの *InputPin* は，それぞれ操作パラメータ param1，param2 に与える引数を表している．

© 2013 Information Processing Society of Japan



(a) 操作の呼び出し　(b) 属性の書き込み　(c) 属性の読み出し

(d) *ObjectFlow*　(d) 変数の書き込み・読み出し

図 6　*Activity* におけるデータの伝搬

Fig. 6 Data propagations in *Activity*.

$$
\begin{aligned}
IN = \{&((n,p),(n',p'))|n,n' \in ObjectNode \wedge p \in (n.properties \cup NULL) \wedge p' \in (n'.properties \cup NULL) \wedge \\
&(n'.properties = \phi \Leftrightarrow p' = NULL) \wedge \\
&(((n.incoming \cap n'.outgoing \neq \phi) \wedge p = p') \vee \\
&(\exists a \in AddStructuralFeatureValueAction\ s.t. \\
&\ (a.result = n \wedge ((a.value = n' \wedge a.structuralFeature = p \wedge p' = NULL) \vee \\
&\quad (a.object = n' \wedge a.structuralFeature \neq p \wedge p = p'))) \vee \\
&(\exists a \in ReadStructuralFeatureAction\ s.t. \\
&\ (a.result = n \wedge (a.object = n' \wedge a.structuralFeature = p' \wedge p = NULL))) \vee \\
&(\exists a \in ReadVariableAction, a' \in AddVariableValueAction\ s.t. \\
&\ ((a,a') \in REACH \wedge a.variable = a'.variable \wedge a.result = n \wedge a'.value = n' \wedge p = p' = NULL)))\}
\end{aligned}
$$

$$
\begin{aligned}
KILL = \{&(a,a')|a \in (AddVariableValueAction \cup RemoveVariableValueAction \cup ClearVariableAction) \wedge \\
&a' \in AddVariableValueAction \wedge a \neq a' \wedge a.variable = a'.variable\}
\end{aligned}
$$

$$
\begin{aligned}
CF = \{&(n,n')|n,n' \in (ControlNode \cup ExecutableNode) \wedge \\
&(\exists e \in ControlFlow\ s.t.\ (n.incoming = n'.outgoing = e) \vee \\
&\ (n,n' \in Action \wedge \exists e \in ObjectFlow\ s.t.\ (e.source \in n'.output \wedge e.target \in n.input)))\}
\end{aligned}
$$

$$
\begin{aligned}
REACH = \{&(n,n') \in CF^{+}|n' \in AddVariableValueAction \wedge (n,n') \notin KILL \wedge \\
&(((n,n') \in CF) \vee \exists (n'',n''') \in REACH\ s.t.\ ((n,n'') \in CF \wedge n' = n''')))\}
\end{aligned}
$$

図 7　*ObjectNode* 間の依存関係 *IN*

Fig. 7 *ObjectNode* dependencies *IN*.

み（*AddStructuralFeatureValueAction*）の場合には，value ノードの値が result ノードの属性 prop に伝搬し，object ノードの prop 以外の属性の値が，result ノードの対応する属性に伝搬する．一方，図 6 (c) に示す属性の読み出し（*ReadStructuralFeatureAction*）の場合には，object ノードの属性 prop の値が result ノードに伝搬する．図 7 の *IN* の定義では，4–6 行目の論理式によって前者の書き込みの場合を，7，8 行目の論理式によって後者の読み出しの場合を，それぞれ表している．

第 3 の伝搬は，変数値の書き込み・読み出しの組合せによって表現される．図 6 (e) に示すように，先行する *Action* で変数 var に value ノード値の書き込み（*AddVariableValueAction*）を行い，後続の *Action* で var の値の読み出し（*ReadVariableAction*）を行う場合に，value ノードの値が変数 var を介して result ノードに伝搬する．図 7 の *IN* の定義では，9，10 行目の論理式を用いてこの種の伝搬を表している．関係 $REACH \subseteq (ControlNode \cup ExecutableNode) \times AddVariableValueAction$ は，*Activity* 中の $n \in ControlNode \cup ExecutableNode$ と，n に到達する変数の書き込みを行う $n' \in AddVariableValueAction$ の組を保持する．

ここで，関係 *REACH* を定義するため，*Activity* の制御の順序関係 $CF \subseteq (ControlNode \cup ExecutableNode) \times (ControlNode \cup ExecutableNode)$ と，変数の無効化関係 $KILL \in (AddVariableValueAction \cup RemoveVariableValueAction \cup ClearVariableAction) \times AddVariableValueAction$ を導入する．$(n,n') \in CF$ は，n が n' の制御上の後続ノードであることを意味する．すなわち n' から n への *ControlFlow* が存在するケース，あるいは $n,n' \in Action$ であり，かつ n' に接続された *OutputPin* から，n に接続された *InputPin* への *ObjectFlow* が存在するケースのいずれかに該当する．また $(a,a') \in KILL$ は，a' における変数の書き込みが，a における書き込みや削除によって無効化されることを意味する．

CF と *KILL* を用いた，関係 *REACH* の定義を図 7 に示す．$(n,n') \in REACH$ となるのは，n の直前ノード n' で変数の書き込みが行われるケース，あるいは n' での書き込みが，n のある直前ノード n'' を介して n に到達するケースのいずれかである．したがって，*REACH* は *CF* の推移閉包を用いて表現される．また n' における変数への書き込みが n で無効化されるケースは，*KILL* を用いた条件式によって除外される．*REACH* の定義には *REACH* 自身が含ま

© 2013 Information Processing Society of Japan

れている．$REACH = \phi$ からスタートして，条件を満たす組 (n, n') を $REACH$ に追加し，不変になるまで繰り返し計算することによって，$REACH$ を求めることができる．

到達定義の解析

$Activity$ の $ObjectNode$ の属性の到達定義 $DEF \subseteq (ObjectNode \times (Property \cup \mathrm{NULL})) \times (ObjectNode \times (Property \cup \mathrm{NULL}))$ を求める．DEF は，$ObjectNode$ の属性の伝搬関係 $PG \subseteq (ObjectNode \times (Property \cup \mathrm{NULL})) \times (ObjectNode \times (Property \cup \mathrm{NULL}))$ の推移関係として定義される．

DEF，PG の定義を**図 8** に示す．$((n, p), (n', p')) \in PG$ は，n' の属性 p' の値が n の属性 p に伝搬することを表す．PG には，$Activity$ 内の依存関係 IN の伝搬（2 行目の条件式）と，操作呼び出しを介した $Activity$ 間のデータ伝搬の 2 通りの伝搬が含まれる．$Activity$ 間のデータ伝搬は，i) n が $Activity$ の入力パラメータを表す $ActivityParameterNode$ であり，n' が入力パラメータに与える実引数を表す $ActionInputPin$ であるケース，ii) n' が $Activity$ の出力パラメータを表す $ActivityParameterNode$ であり，n が出力パラメータの戻り値を表す $OutputPin$ であるケース，の 2 通りのケースで発生する．図 8 の PG の定義では，i) のケースを 3，4 行目の条件式で，ii) のケースを 5，6 行目の条件式で，それぞれ表している．

図 1 の例題モデルに対して本ステップのデータフロー解析を適用すると，**表 2** の $KILL$，CF，$REACH$，IN，DEF が得られる．ただし表 2 ではスペースの都合上，DEF については，Step 4 の整合性検証の際に利用する，番号 6，35 のノードに関連する一部の情報のみ，コンパクトな $DEF_{(n,p)}$ の形式で記載している．$DEF_{(n,p)}$ は $(n, p) \in ObjectNode \times (Property \cup \mathrm{NULL})$ に対して定義され，$DEF_{(n,p)} = \{(n', p') \in ObjectNode \times (Property \cup \mathrm{NULL}) | ((n, p), (n', p')) \in DEF\}$ である．

3.4 Step 4：整合性検証

本ステップでは，あるシステム機能に関する各参照 $r \in REF$ の整合性を判定する．まず，破壊的アクセスと補完的アクセスの操作呼び出しの履歴を表すアクセス履歴 $HIST$ と，呼び出し対象の操作の種別を表すアクセス種別 $ACTYPE$ を計算する．そして，これらの情報を利用して $Activity$ の検証を行う．なお以下では，3.2 節で述べた実行パターン (1) と (1-1)，(1-2)，(1-3) の組合せについて説明するが，(2) と (2-1)，(2-2) の組合せについても同様の手順で検証を行うことができる．

アクセス履歴とアクセス種別の計算

アクセス履歴 $HIST \subseteq REF \times (ControlNode \cup ExecutableNode) \times CallOperationAction$ と，アクセス種

$$
\begin{aligned}
&DEF = PG^{+}\\
&PG = \{((n,p),(n',p')) | n, n' \in ObjectNode \wedge p \in (n.properties \cup NULL) \wedge p' \in (n'.properties \cup NULL) \wedge\\
&\qquad (((n,p),(n',p')) \in IN \vee\\
&\qquad (n \in ActivityParameterNode \wedge n.parameter.direction = in \wedge n'.name = n.name \wedge\\
&\qquad \exists c \in CG\ s.t.\ (c.op = n.activity.specification \wedge n' \in c.a'.input) \wedge p' = p) \vee\\
&\qquad (n' \in ActivityParameterNode \wedge n'.parameter.direction = out \wedge n.name = n'.name \wedge\\
&\qquad \exists c \in CG\ s.t.\ (c.op = n'.activity.specification \wedge n \in c.a'.output) \wedge p = p'))\}
\end{aligned}
$$

図 8 到達定義 DEF

Fig. 8 Reaching definitions DEF.

表 2 例題モデルの解析結果

Table 2 The analysis result of example model.

集合	値
$KILL$	ϕ
CF	{(3,1),(4,3),(7,4),(9,7),(10,9),(19,10),(12,9),(16,12),(17,16),(19,17),(25,24),(29,25),(33,29),(36,33),(37,36)}
$REACH$	{(4,3),(7,3),(9,3),(10,3),(12,3),(16,3),(17,3),(19,3)}
IN	{((2,NULL),(20,NULL)),((6,NULL),(5,NULL)),((15,NULL),(13,NULL)),((14,NULL),(21,NULL)), ((23,NULL),(11,NULL)),((22,NULL),(18,NULL)),((28,url),(26,url)),((28,id),(26,id)),((27,NULL), (38,NULL)),((32,url),(30,url)),((32,id),(30,id)),((31,NULL),(39,NULL)),((35,url),(34,url)), ((35,id),(34,id)),((30,id),(28,id)),((30,url),(27,NULL)),((34,url),(32,url)),((34,id),(31,NULL)), ((5,NULL),(2,NULL)),((13,NULL),(2,NULL))}
$DEF_{(6,\mathrm{NULL})}$	{(5,NULL),(2,NULL),(20,NULL)}
$DEF_{(35,\mathrm{id})}$	{(34,id),(31,NULL),(39,NULL),(15,NULL),(13,NULL),(2,NULL),(20,NULL)}
$ACTYPE$	{(r_0,7,(wsDefRegistered,WSDef,id,wsDefID,NULL,return,NULL)), (r_0,36,(addLocator,WSLocator,wsDefID,locator,id))}
$HIST$	{(r_0,7,7),(r_0,9,7),(r_0,10,7),(r_0,19,7),(r_0,12,7),(r_0,16,7),(r_0,17,7),(r_0,24,7),(r_0,25,7),(r_0,29,7), (r_0,33,7),(r_0,36,7),(r_0,37,7),(r_0,36,36),(r_0,37,36),(r_0,16,36),(r_0,17,36),(r_0,19,36)}

© 2013 Information Processing Society of Japan

$$
\begin{aligned}
HIST = \{(r,n,n') | & r \in REF \wedge n \in (ControlNode \cup ExecutableNode) \wedge n' \in CallOperationAction \wedge \\
& \exists e \in (A_r^{src} \cup G_r^{dst} \cup C_r^{dst} \cup A_r^{dst})\ s.t.\ e.op = n'.operation \wedge \\
& (n = n' \vee \\
& \exists n'' \in (ControlNode \cup ExecutableNode)\ s.t.\ (r,n'',n') \in HIST \wedge \\
& ((n,n'') \in CF \vee \\
& (n \in InitialNode \wedge \exists c \in CG\ s.t.\ (c.op = n.activity.specification \wedge c.a' = n''))))\}
\end{aligned}
$$

$$
\begin{aligned}
ACTYPE = \{(r,n,e) | & r \in REF \wedge n \in CallOperationAction \wedge \\
& e \in (A_r^{src} \cup G_r^{dst} \cup C_r^{dst} \cup A_r^{dst}) \wedge e.op = n.operation\}
\end{aligned}
$$

図 9 アクセス履歴 *HIST* とアクセス種別 *ACTYPE*

Fig. 9 Access history *HIST* and access type *ACTYPE*.

```
VERIFY(REF, AC, DEF, HIST, ACTYPE)
1  foreach r ∈ REF
2    foreach n ∈ (ControlNode ∪ ExecutableNode)
3      foreach h ∈ HIST s.t. (h.r = r ∧ h.n = n)
4        foreach t ∈ ACTYPE s.t. (t.r = r ∧ t.n = h.n' ∧ t.e ∈ A_r^src)
5          d_reach := {(o', p') ∈ ObjectNode × (Proeprty ∪ NULL)|
6            ∃((o, p), (o', p')) ∈ DEF s.t. o ∈ t.n.input ∧ o.name = t.e.param.name ∧ p = t.e.prop_param}
7          c_reach := φ
8          foreach h' ∈ HIST s.t. (h'.r = r ∧ h'.n = n ∧ h'.n' ≠ h.n')
9            foreach t' ∈ ACTYPE s.t. (t'.r = r ∧ t'.n = h'.n')
10             if t'.e ∈ G_r^dst
11             then c_reach := c_reach ∪ {(o', p') ∈ ObjectNode × (Proeprty ∪ NULL)|
12               o' ∈ t'.n.output ∧ o'.name = t'.e.param.name ∧ p' = t'.e.prop_param}
13             if t'.e ∈ C_r^dst
14             then c_reach := c_reach ∪ {(o', p') ∈ ObjectNode × (Proeprty ∪ NULL)|
15               ∃((o, p), (o', p')) ∈ DEF s.t. o ∈ t'.n.input ∧ o.name = t'.e.param_in.name ∧ p = t'.e.prop_in}
16             if t'.e ∈ A_r^dst
17             then c_reach := c_reach ∪ {(o', p') ∈ ObjectNode × (Proeprty ∪ NULL)|
18               ∃((o, p), (o', p')) ∈ DEF s.t. o ∈ t'.n.input ∧ o.name = t'.e.param.name ∧ p = t'.e.prop_param}
19         if (c_reach ∩ d_reach) = φ then output 'INCORRECT'
```

図 10 整合性の検証アルゴリズム

Fig. 10 The algorithm for verifying *Activities*.

別 $ACTYPE \subseteq REF \times CallOperationAction \times (A \cup G \cup C)$ を求める．*HIST*，*ACTYPE* の定義を図 9 に示す．

$(r,n,n') \in HIST$ は，*Activity* 中の $n \in ControlNode \cup ExecutableNode$，あるいは n の制御フロー上の祖先ノード n' において，$r \in REF$ に関する破壊的アクセスまたは補完的アクセスの操作呼び出しが行われることを意味する．図 9 の *HIST* の定義では，3 行目の条件式によって $n = n'$ のケースを表し，4–6 行目の条件式によって，*Activity* 内または *Activity* 間にまたがる制御フロー上の先行ノード n'' から，アクセス履歴が引き継がれることを表している．*HIST* の定義には *HIST* 自身が含まれるため，*REACH* と同様に繰り返し計算を行い，*HIST* を不動点として求めることができる．

$(r,n,e) \in ACTYPE$ は，$n \in CallOperationAction$ において，$r \in REF$ に関する破壊的アクセスまたは補完的アクセス e の操作呼び出しが行われることを意味する．

整合性判定

各参照 $r \in REF$ の整合性を判定するアルゴリズム VERIFY を図 **10** に示す．Step 2 で述べたように，参照整合性の破壊を防ぐために，補完的アクセスは破壊的アクセスに先立って実行される．そこで，*Activity* の各 $n \in ControlNode \cup ExecutableNode$ のアクセス履歴 $h \in HIST$ の操作呼び出し $h.n'$ が，r に関する破壊的アクセス $t.e \in A_r^{src}$ の操作呼び出しである場合[*4,*5]に（1–4 行目），n の他のアクセス履歴 $h' \in HIST$ の操作呼び出し $h'.n'$ が，$h.n'$ による破壊を防ぎうることを基準として，整合性を判定する（5–19 行目）．

ここで 3.2 節で述べた実行パターンの説明から，$h'.n'$ が $h.n'$ による破壊を防ぐ条件は，$h'.n'$ が補完的アクセス $t'.e \in G_r^{dst} \cup C_r^{dst} \cup A_r^{dst}$ を呼び出し[*4,*5]，かつ以下に定義する $v_{h'.n'}$ が，$h.n'$ の，パラメータ $t.e.param$ を表す $o_{h.n'} \in InputPin$ の属性 $t.e.prop_{param}$ の値 $v_{h.n'}$ と等しいことである．

$$
v_{h'.n'} = \begin{cases}
h'.n' \text{ の}, t'.e.param \text{ を表す } o_{h'.n'} \in InputPin \cup OutputPin \text{ の，属性 } t'.e.prop_{param} \text{ の値} \\
\quad (t'.e \in G_r^{dst} \cup A_r^{dst} \text{ の場合}) \\
h'.n' \text{ の，} t'.e.param_{in} \text{ を表す } o_{h'.n'} \in InputPin \text{ の，属性 } t'.e.prop_{in} \text{ の値} \\
\quad (t'.e \in C_r^{dst} \text{ の場合})
\end{cases}
$$

すなわち，$\forall h\ \exists h'\ s.t.\ (h \neq h' \wedge (v_{h.n'} = v_{h'.n'}))$ が検証

*4 変数 $t, t' \in ACTYPE$ は，$h.n'$，$h'.n'$ に対応するアクセス種別として，図 10 の 4，9 行目でそれぞれ定義されている．

*5 VERIFY で参照している A_r^{src}，G_r^{dst}，C_r^{dst}，A_r^{dst} は，参照 r とアクセス情報 *AC* から，表 1 の内容に従って計算される．

© 2013 Information Processing Society of Japan

の判定基準である．

上記の同値関係 $v_{h.n'} = v_{h'.n'}$ の成立は，*Activity* の制御とデータの両方のフローに依存するため，静的な解析によって厳密に判定することは困難である．そこで本手法では，基準を緩和して，データフローに基づく次の 2 つの条件の成立を判定する．これらの条件が満たされるならば，データの依存関係により，同値関係 $v_{h.n'} = v_{h'.n'}$ が成立する可能性がある．

条件 1： $t'.e \in G_r^{dst}$ のとき，$o_{h'.n'}$ の属性 $t'.e.prop_{param}$ から $o_{h.n'}$ の属性 $t.e.prop_{param}$ へのデータ伝搬が存在する．すなわち $o_{h.n'}$ の属性 $t.e.prop_{param}$ の到達定義に，$o_{h'.n'}$ の属性 $t'.e.prop_{param}$ が含まれる．

条件 2： $t'.e \in A_r^{dst}$（$t'.e \in C_r^{dst}$）のとき，$o_{h.n'}$ の属性 $t.e.prop_{param}$ と $o_{h'.n'}$ の属性 $t'.e.prop_{param}$（$t'.e.prop_{in}$）に共通のデータ伝搬元 $(o,prop) \in ObjectNode \times Property$ が存在する．すなわち，これらの $o_{h.n'}$ の属性と $o_{h'.n'}$ の属性の共通定義が存在する．

図 10 のアルゴリズムは，Step 3 で求めた *DEF* から，$o_{h.n'}$ の属性 $t.e.prop_{param}$ の到達定義を d_{reach} として求め（5，6 行目），各 $o_{h'.n'}$ の属性と，その到達定義の和集合を c_{reach} として求める（7–18 行目）．c_{reach} と d_{reach} が互いに素のときに，検証失敗と判定する（19 行目）．

図 1 の例題モデルの唯一のシステム機能に対して，アクセス履歴とアクセス種別を計算すると，表 2 の *HIST* と *ACTYPE* が得られる．表 2 の *ACTYPE* の値と，Step 2 の説明で述べた $A_{r_0}^{src}$，$C_{r_0}^{dst}$ の値から，番号 7，36 のノードで，それぞれ r_0 の補完的アクセス $(\mathrm{wsDefRegistered}, \mathrm{WSDef}, \mathrm{id}, \mathrm{wsDefID}, \mathrm{NULL}, \mathrm{return}, \mathrm{NULL}) \in A_{r_0}^{src}$，破壊的アクセス $(\mathrm{addLocator}, \mathrm{WSLocator}, \mathrm{wsDefID}, \mathrm{locator}, \mathrm{id}) \in C_{r_0}^{dst}$ の操作呼び出しが行われることが分かる．

表 2 に対するアルゴリズム VERIFY の判定結果について以下に説明する．*HIST* の要素の中で，VERIFY の 5 行目以降の判定の対象となるのは，破壊的アクセスの操作呼び出しの履歴を表す $(r_0, 36, 36)$，$(r_0, 37, 36)$，$(r_0, 16, 36)$，$(r_0, 17, 36)$，$(r_0, 19, 36)$ の 5 つである．これらのいずれについても $d_{reach} = DEF_{(35,\mathrm{id})}$ であり，また対応する補完的アクセスの操作呼び出しの履歴を表す $(r_0, 36, 7)$，$(r_0, 37, 7)$，$(r_0, 16, 7)$，$(r_0, 17, 7)$，$(r_0, 19, 7)$ から $c_{reach} = DEF_{(6,\mathrm{NULL})}$ となる．$d_{reach} \cap c_{reach} = \{(2, \mathrm{NULL}), (20, \mathrm{NULL})\} \neq \phi$，すなわち番号 35 のノードの属性 id と，番号 6 のノードには共通の定義が存在し，判定の条件 2 が成り立つため，アルゴリズムは検証成功と判定する．

4. 議論

本章ではまず，提案手法の妥当性と検証能力を把握し，手法の利用方法について述べる．さらに，実開発における手法の有用性について論じる．

4.1 手法の妥当性と検証能力

技術課題 1–3 に対応して，破壊的アクセス・補完的アクセスの定義，アクセス操作に与える引数の由来，アクセス順序を考慮した *Activity* の検証，の 3 つの観点から手法の妥当性と検証能力を考察した後，本手法の有効な利用方法を述べる．

破壊的アクセス・補完的アクセスの定義

関係データベースにおける参照整合性と，整合性を確保するために一般的に用いられる実現方法の考え方は，DC 間にまたがる参照の問題に対しても自然に適合する．3.2 節で述べた我々の方法は，これらの確立された技術の類比に基づいて，破壊的アクセス・補完的アクセスの組合せを定義しており，妥当な方法であるといえる．

アクセス操作に与える引数の由来

3.3 節で導入した *Activity* のデータフロー解析方法は，Field-sensitive であり，アクセス操作の呼び出しに与える引数またはその属性の由来，すなわち到達定義を，網羅的に抽出する能力を持つ．ただし，Step 3 における到達定義 *DEF* の解析は，制御の分岐条件の評価を実施しない Path-insensitive な方法であるため，実際には到達しない定義が，解析結果に含まれる可能性がある．また本論文では，操作の呼び出し箇所の文脈を考慮せずに *Activity* 間にまたがるデータ伝搬を解析する Context-insensitive な方法を述べたが，呼び出し箇所の文脈を考慮する Context-sensitive な方法に変更することは容易である．

アクセス順序を考慮した *Activity* の検証

3.4 節で説明したアクセス履歴 *HIST* の計算は，分岐条件の評価を実施しない Path-insensitive な方法であり，制御フロー上のノード間で，アクセス操作の呼び出し履歴を過剰に引き継ぐ可能性がある．ただし破壊的アクセスと補完的アクセスの呼び出し履歴を一律に引き継ぐため，アクセス履歴の計算結果が整合性の判定誤りを招く恐れはない．

3.4 節の整合性検証のアルゴリズム VERIFY は，破壊を防止する補完的アクセスの呼び出し記述の欠落（欠陥 1），破壊的アクセス・補完的アクセスの操作に不適切な引数を与える欠陥（欠陥 2），補完的アクセスよりも先に破壊的アクセスを呼び出す欠陥（欠陥 3）の 3 種類の欠陥を検出することができる．VERIFY は，アクセス操作に与える引数の厳密な同値判定を行う代わりに，データフローに基づく緩和された基準を用いて，同値関係の成立の可能性を判定する．制御フローが同値関係に影響を与える場合には，検証結果に偽陰性と偽陽性の両方の誤りが生じうる．前者の場合，VERIFY の出力結果が検証成功であっても，*Activity* の組合せによって参照整合性が破壊される．後者の場合は，VERIFY の出力結果が検証失敗であっても，実際には参照整合性が保たれる．

検証結果の偽陽性の誤りの一例を示す．ある *Activity* の中で，まず補完的アクセス $t'.e \in G_r^{dst}$ を呼び出し，

$t'.e.prop_{param}$ を介して参照先の属性 $t'.e.prop_{cls}$ の値集合 V を取得する．次に，変数 x の値の V に対する帰属関係を判定し，判定結果が真の場合に，破壊的アクセス $t.e \in A_r^{src}$ を呼び出して，引数の属性 $t.e.prop_{param}$ に x の値を与える．この記述では，変数 x の V に対する帰属関係の判定によって，$t'.e.prop_{param}$ の値が $t.e.prop_{param}$ に渡されることが制御フロー上保証される．しかし本手法のデータフロー解析では分岐条件件の評価を行わないため，この同値関係を検出できず，アルゴリズムは誤って検証失敗と判定する．

本手法の利用方法

本手法の利用者は，手法の入力として，2 章で説明した仕様モデルを用意する必要がある．構造モデルを表すクラス図では，図 1 の例のように，ステレオタイプ ⟨⟨SC⟩⟩，⟨⟨DC⟩⟩ を用いて 2 種類のコンポーネントを区別し，UML の依存（*Dependency*）を用いて参照関係 *REF* を記述する．UML では DC のアクセス情報 *AC* を記述するための図や記法が定義されていないため，*AC* の内容はテキスト形式で記述する．また命名規則やステレオタイプの付与等の手段を用いて，システム機能を提供する SC の操作を特定する．

本手法では UML のメタモデルに基づいて，仕様モデルの解析を行う．したがって，仕様モデルの内容は UML のメタモデルに準拠している必要がある．また，検証対象のシステム内の一部のコンポーネントの仕様の情報が不足していると，十分な検証結果が得られない可能性があるため，システムに含まれるすべての SC，DC の仕様を入力として与える必要がある．

本手法が検証失敗を出力した場合に，検証対象である *Activity* の組合せに欠陥 1–3 が存在する可能性がある．対象をより詳細に調べることによって，欠陥の発見と修正につながるケースがありうる．したがって，*Activity* 中の *Action* 記述の欠陥を探索する用途で，本手法を有効に利用することができる．その際に，検証結果に含まれる真の欠陥と，前述の偽陽性の判定誤りを，人間の判断によって判別することが必要である．

一方，本手法が検証成功と判定した場合は，欠陥 1–3 が発見されなかったことを意味するが，参照整合性が保証されるわけではない．参照整合性を保証するためには，本手法の検証対象である *Action* の記述に加えて，制御フローの検証が必要である．たとえば図 1 において，手順 E で破壊的アクセスの呼び出しを行う場合には，手順 B の補完的アクセスの照合結果が true でなければならない．しかし照合結果の判定は，*Action* ではなく制御を表す *ControlNode* で記述されており，本手法では検証することができない．また前述のように，本手法の検証結果には潜在的に偽陰性の判定誤りが含まれる．すなわち，参照整合性を証明する用途で本手法を適用することは不適切である．

4.2 手法の有用性

我々は，既存の UML の処理系 [9] を利用して，本手法の解析を自動化するツールを作成した．そのツールを，筆者の勤務先の社内システムに適用して，手法の有用性を確認する実験を行った．このシステムは，Web ブラウザを介してシミュレーションを投入する機能を提供する．シミュレーションは WebService で定義され，計算ノード上で実行される．利用者は WebService の実行手順を BPEL フローとして記述し，システムに投入する．

対象システムは 5 つの DC と 7 つの SC から構成される．DC には計 58 のデータアクセス操作が定義され，$|A| = 42$，$|G| = 91$，$|D| = 6$，$|C| = 44$ である．DC 間には 25 の参照が存在する．すなわち $|REF| = 25$ である．一方 SC には，システム機能を提供する 35 のシステム操作と，システム操作から利用される 13 の内部操作が定義されている．本実験では，これらの SC の操作の振舞いを記述する 48 個の *Activity* の検証を行った．検証対象の *Activity* 集合の規模と，*Activity* のノード種別ごとの出現頻度を**表 3**，**表 4** に示す．

実験では，まず人手により，参照整合性の観点から仕様モデルのレビューを行い，レビューで発見した欠陥を修正する．その後，修正済みのモデルをツールを用いて自動で検証した．一般的な能力（Intel Core2 Duo 1.60 GHz，4 GB RAM）を持つ PC を用いて検証を行ったところ，ツールの実行時間は 833 ms であった．

自動検証の結果を**表 5** に示す．ツールは，対象システムが提供する 35 のシステム操作について，27 操作を検証成功，8 操作を検証失敗と判定した．検証失敗のケースを詳細に調べたところ，7 個の欠陥と，4 個の判定誤りが含まれていた．ただし，いくつかの検証失敗のケースは，複数の欠陥あるいは判定誤りに帰する．7 個の欠陥には，4.1 節で述べた 3 種類の欠陥（欠陥 1–3）が含まれていた．また 4 個の判定誤りは，いずれも 4.1 節で述べた，制御フローに基づく偽陽性の誤りであった．

7 個の欠陥の具体的な記述内容は欠陥ごとに異なるが，欠陥を含むケースのコールグラフ *CG* の平均サイズは 13.9 であり，全体の平均値 5.7（表 3）よりも大きかった．一般に，*Activity* の整合性を人手で確認する場合，確認対象の *Activity* の数が増えるほど，見落とし等の誤りが発生しやすいと考えられる．本手法の網羅的な自動検証は，人間が苦手とする，多数の *Activity* にまたがる確認を行う場合に

表 3 *Activity* 集合の規模
Table 3 The dimensions of *Activities*.

項目	平均値	最大値
コールグラフ CG のサイズ	5.7	31
Activity のノード数	35.2	103
Activity のエッジ数	23.3	77

© 2013 Information Processing Society of Japan

表 4 *Activity* ノードの出現頻度（百分率）
Table 4 The frequency of *Activity* node types in percent values.

ControlNode		*ObjectNode*		*ExecutableNode*	
InitialNode	3.3	*InputPin*	36.1	*StructuredActivityNode*	0.6
FlowFinalNode	0.6	*OutputPin*	16.8	*(Create/Destroy) ObjectAction*	0.7
ActivityFinalNode	3.7	*ActivityParameterNode*	12.5	*(Add/Remove) StructuralFeatureValueAction*	1.5
Decision/Merge Node	4.7	*ExpansionNode*	0.6	*(Clear/Read) StructuralFeatureAction*	3.5
Fork/Join Node	0.0			*(Add/Remove) VariableValueAction*	0.7
				(Clear/Read) VariableAction	0.5
				ValueSpecificationAction	2.5
				CallOperationAction	11.7

表 5 ツールを用いた自動検証の結果
Table 5 The result of automated verification.

検証成功	検証失敗				
		補完的アクセスの欠落	データ伝搬の欠陥	アクセス順番の欠陥	判定誤り
27	8	5	1	1	4

おいて，特に有用である．

本手法の適用により，人手のレビューでは検出できなかった 7 個の欠陥を発見することができた．また，5 つのシステム操作の検証には，Planas らが定義した *Action* 記述列では表現できない，複数の *Action* を含むループを有する *Activity* の解析が必要であった．本手法で技術課題 1–3 を解決することで，これらの検証と欠陥の発見が可能となった．

以上のように，本実験を通じて，技術課題 1–3 の実用上の重要性と，提案手法の有効性を確認することができた．本手法を用いて，人手で除去しきれなかった *Activity* の欠陥を取り除くことによって，仕様定義段階で参照整合性を確保することが可能になる．また本実験の実施順番と逆に，ツールを用いて自動検証を実行した後に，その検証結果を参照して仕様レビューを行うことによって，人手による欠陥の発見・修正の正確性と効率の向上が期待できる．

5. 関連研究

本論文では，Planas らの補完 *Action* のアイデア [5], [6] を複数の DC 間に拡張するために，*Activity* の Field-sensitive なデータフロー解析の方法を導入した．従来，*Activity* のデータフロー解析に関して，Meda らがデータフロー方程式を用いた方法 [10] を，Storrle がカラーペトリネットに基づく方法 [11] を，それぞれ提案している．また Waheed らは文献 [12] において，*Action* 記述を含む UML の状態遷移モデルのデータフロー解析方法を提案している．しかしこれらの方法は，いずれもオブジェクトの属性値の伝搬を考慮しない，Field-insensitive な方法である．また，C 言語等の型のキャストを含むプログラムの解析では，キャストの際の構造体の属性値の伝搬の追跡が課題となる．この課題に対処するため，Wilson らは文献 [13] において，構造体の属性のオフセットを利用して解析を行う方法を提案している．UML でも *ReclassifyObjectAction* を用いてオブジェクトの型をキャストすることが可能であるが，本手法ではこの *Action* を扱わないため，上記のキャストに起因する課題は生じない．

さらに本論文では，*Action* の順序関係を考慮した *Activity* の検証の問題に取り組んだ．プログラム解析の分野における類似の問題として，プログラム中のイベント記述の順序制約の検証があげられる．Olender らは文献 [14] において，手続内のイベント記述と順序制約の整合性を静的に検証する手法を提案し，文献 [15] において，手法の適用範囲を手続間に拡張した．Olender らの手法では，順序制約を有限オートマトンとして表現し，プログラム終了時にオートマトンがとりうる状態を解析することによって，制約の検証を行う．この方法は，オートマトンとして表現される任意の順序制約を汎用的に検証することができる．一方，本論文で提案した整合性判定のアルゴリズム VERIFY は，補完的アクセスが破壊的アクセスに先立って実行される点のみを検証する，専用アルゴリズムである．複雑な *Action* の順序関係を検証する場合には，Olender らの汎用手法の適用を検討する価値がある．

モデル駆動開発における実行可能モデルの記述に *Action* が用いられる．OMG は UML の実行可能なサブセットとして fUML を定めた [16]．fUML の *Action* には，変数を扱わない，生成オブジェクトの型を *Class* に限定する，等の制限がある．しかし，本論文で提案した Field-sensitive なデータフロー解析と，アクセス操作の呼び出し履歴の解析の方法は，UML の *Action* に対して設計されており，fUML に基づく *Activity* にも適用可能である．

6. おわりに

本論文では，UML の *Activity* を用いて記述される，サービスコンポーネントの振舞いを静的に解析して，データコンポーネント間にまたがる参照整合性を自動で検証する手法を提案した．本手法で用いるデータフロー解析は Field-sensitive であり，*Activity* 中のオブジェクトの属性の

© 2013 Information Processing Society of Japan

伝搬を解析する．また本手法の検証アルゴリズムは，データアクセス操作を呼び出す *Action* の順序関係を考慮して，*Activity* の検証を行う．さらに本論文では，実問題の仕様モデルを用いて，提案手法が，人手で除去しきれない *Activity* の欠陥を取り除くうえで有用であることを確認した．

本手法を用いて，*Activity* の *Action* 記述の欠陥を発見・修正し，仕様定義段階で参照整合性を確保することが可能になる．今後は，多くの実開発に手法の導入を図るとともに，手法の拡張と応用について検討していきたい．

参考文献

[1] D'Souza, D. and Wills, A.: *Objects, Components and Frameworks With UML: The Catalysis Approach*, Addison-Wesley (1998).

[2] Cheesman, J. and Daniels, J.: *UML Components – A Simple Process for Specifying Component-Based Software*, Addison-Wesley (2001).

[3] Trčka, N., van der Aalst, W. and Sidorova, N.: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows, *Advanced Information Systems Engineering*, van Eck, P., Gordijn, J. and Wieringa, R. (Eds.), Lecture Notes in Computer Science, Vol.5565, pp.425–439, Springer, Berlin/Heidelberg (2009).

[4] Ryndina, K., Kuster, J. and Gall, H.: Consistency of Business Process Models and Object Life Cycles, *Models in Software Engineering*, Kuhne, T. (Ed.), Lecture Notes in Computer Science, Vol.4364, pp.80–90, Springer, Berlin/Heidelberg (2007).

[5] Planas, E., Cabot, J. and Gomez, C.: Verifying Action Semantics Specifications in UML Behavioral Models, *Advanced Information Systems Engineering*, van Eck, P., Gordijn, J. and Wieringa, R. (Eds.), Lecture Notes in Computer Science, Vol.5565, pp.125–140, Springer, Berlin/Heidelberg (2009).

[6] Planas, E., Cabot, J. and Gomez, C.: Lightweight Verification of Executable Models, *Conceptual Modeling – ER 2011*, Jeusfeld, M., Delcambre, L. and Ling, T.-W. (Eds.), Lecture Notes in Computer Science, Vol.6998, pp.467–475, Springer, Berlin/Heidelberg (2011).

[7] Ackermann, J. and Turowski, K.: A Library of OCL Specification Patterns for Behavioral Specification of Software Components, *CAiSE 2006*, Dubois, E. and Pohl, K. (Eds.), LNCS, Vol.4001, pp.255–269, Springer (2006).

[8] Inoue, T. and Honiden, S.: A method for data-flow analysis of business components, *Proc. 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, *CBSE '11*, New York, NY, USA, ACM, pp.51–60 (2011).

[9] MDT project: MDT/UML2, available from ⟨http://www.eclipse.org/modeling/mdt/?project=uml2⟩.

[10] Meda, H.S., Sen, A.K. and Bagchi, A.: On Detecting Data Flow Errors in Workflows, *J. Data and Information Quality*, Vol.2, No.1, pp.4:1–4:31 (2010).

[11] Storrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities, *Electronic Notes in Theoretical Computer Science*, Vol.127, No.4, pp.35–52 (2005).

[12] Waheed, T., Iqbal, M. and Malik, Z.: Data Flow Analysis of UML Action Semantics for Executable Models, *Model Driven Architecture – Foundations and Applications*, Schieferdecker, I. and Hartman, A. (Eds.), Lecture Notes in Computer Science, Vol.5095, pp.79–93, Springer, Berlin/Heidelberg (2008).

[13] Wilson, R.P. and Lam, M.S.: Efficient context-sensitive pointer analysis for C programs, *Proc. ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, *PLDI '95*, New York, NY, USA, ACM, pp.1–12 (1995).

[14] Olender, K.M. and Osterweil, L.J.: Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation, *Engineering of Software*, Tarr, P.L. and Wolf, A.L. (Eds.), pp.115–141, Springer, Berlin/Heidelberg (2011).

[15] Olender, K.M. and Osterweil, L.J.: Interprocedural static analysis of sequencing constraints, *ACM Trans. Softw. Eng. Methodol.*, Vol.1, No.1, pp.21–52 (1992).

[16] OMG: Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0 (2011), available from ⟨http://www.omg.org/spec/FUML/1.0/⟩.

井上 拓　（正会員）

1974 年生．1997 年早稲田大学理工学部応用物理学科卒業．1999 年同大学大学院理工学研究科修士課程修了．同年キヤノン株式会社入社．デジタルカメラの開発，ソフトウェア自動テスト環境の構築に従事．現在，同社デジタルシステム開発本部にて医療機器の開発に従事．要求分析，形式手法，コンポーネントベース開発等に興味を持つ．

本位田 真一　（フェロー）

1953 年生．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，2012 年より同研究所副所長を併任，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．現在，電気通信大学，英国 UCL 等の客員教授を兼任．2005 年度パリ第 6 大学招聘教授．工学博士（早稲田大学）．1986 年度情報処理学会論文賞受賞．日本ソフトウェア科学会理事，情報処理学会理事，日本ソフトウェア科学会編集委員長を歴任．ACM 日本支部会計幹事，日本学術会議連携会員．

© 2013 Information Processing Society of Japan