東京大学
情報理工学系研究科 創造情報学専攻
博士論文

ビジネスコンポーネントに跨る参照整合性のための UML/OCL 振舞いモデルの静的解析の研究

Static Analysis of UML/OCL Behavioral Models for Referential Integrity between Business Components

井上 拓
Taku Inoue

指導教員　本位田 真一 教授

2013 年 6 月

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，データの一貫性の保たれた高品質のソフトウェアを効率的に開発するために，ソフトウェアを構成するビジネスコンポーネントの仕様モデルの整合性を確保することが重要である．しかし，宣言的に記述された仕様モデルについて，モデルの記述者の意図と利用者の解釈を一致させることは容易ではない．また，大規模なシステムの開発では，合成された多数のコンポーネントの仕様モデルが，システムに要求される性質を満たすことを，人手によって正確かつ網羅的に確認することは困難である．本論文では，ビジネスコンポーネントに跨る参照整合性を確立するための，コンポーネントの振舞いモデルの整合性の判定方法の研究を行う．

本研究では，コンポーネントの振舞いモデルの整合性判定を支援するために，2 つの解析手法を定める．これらの解析手法は，振舞いモデルを静的に解析して，整合性の判定を容易に行うための情報を提供する．

第 1 の解析手法は，宣言的な仕様記述言語である OCL（Object Constraint Language）で記述された振舞いモデルが表すデータアクセスを抽出する．この解析手法は，抽象解釈技術を用いて OCL 記述の意味の近似的な解釈を行う．OCL 記述に対する抽象解釈の適用は，従来の研究と異なる，新しいアプローチである．本解析の出力を用いて，データアクセスに関する，モデル記述者の意図と利用者の解釈の整合性を判定することが可能になる．

第 2 の解析手法は，UML Activity で記述された振舞いモデルの組み合わせにおける，データアクセス操作の利用手順の整合性を判定する．この解析手法は，参照整合性を破壊しうるデータアクセスと，破壊を防ぐために必要なデータアクセスの組を定義し，Activity の属性依存のデータフロー解析を行う．これらは，UML Activity の組合せを対象として，関係データベースの参照整合性の実現方法と，データフロー解析の技術を応用したものであり，従来にない新しい応用の形態である．さらに，解析手法は，データアクセス操作の引数の到達定義と，操作呼び出しの履歴情報を利用して，データアクセス手順の整合性の判定を行う．

2 つの手法はいずれも静的な解析方法であり，仕様定義段階でソフトウェアを動作させずに整合性を判定することができる．本研究では，解析の実行を自動化する支援ツールの開発を行った．ツールを自動実行して，その出力情報を用いて整合性の判定を行うことにより，判定の正確性と効率の向上が可能になる．

本論文では，2 つの解析手法の提案を行い，アプローチの妥当性や手法の能力について議論する．さらに，実システムの開発に対して実施した評価実験の結果を報告し，実開発における手法の有用性を確認する．

Abstract

Nowadays, component-based software development (CBD) is widely considered as an effective and efficient approach to construct information systems; so as to ensure system quality in CBD, it is crucial to maintain the consistency of specification models of composing business-components with regards to system requirements at design-time. However, as for the declarative specification models, differences in interpretation of models are apt to occur between modeler and users. Also, systems are generally composed of a number of components, therefore it is a difficult task to verify the consistency of component models accurately and completely by hand. In this thesis, the author tackles the problem of verifying consistency of component behavioral models with regards to referential integrities across business-components, and proposes two different static analysis methods as solutions to the problem.

The first method analizes the behavioral models written in OCL (Object Constraint Language), a declarative modeling language, and identifies the data access described in the models. The method employs an abstract interpretation in order to gain the meaning of OCL description; applying the abstract interpretation technique to OCL is an unconventional novel approach. With the method, the human interpretations of data-access described in the models can be verified.

The second method analizes the behavioral models of UML Activities to verify the calling sequence of data-access operations in Activities. On the analogy of Relational Database Management Systems, the method defines a data access protocol, namely a pair of violative/complementary data access, and validate Activities' conformance with the protocol. The method also performs field-sensitive data-flow analysis of Activities to obtain the reaching definitions of calling parameters passed to data access operations, which are used in the protocol validation. These analysis techniques, to the best of the author's knowledge, have not been applied to UML Activity.

In both methods the analysis is performed by automated tools statically without actually executing models or programs, hence the methods make the accurate and complete verification in upstream process feasible. In the thesis, the author discusses the validity and capabilities of the proposed methods and evaluates methods' usefulness based on the experiment with a real life system.

謝辞

本博士研究を御指導頂いた東京大学大学院及び国立情報学研究所 本位田真一 教授に深謝の意を表します．本位田教授には，筆者がソフトウェア工学の教育プロジェクトであるトップエスイーを受講していたときに，ソフトウェア工学の研究に取り組むきっかけを与えて頂きました．そして，博士課程の研究では，筆者が選んだ研究テーマを追究する自由を与えていただくと同時に，研究生活の厳しい局面において，数々の貴重な御助言と御指導をいただきました．本位田教授の御指導の下で学んだ様々な教えを，今後の人生に生かしていきたいと考えています．

本研究の論文審査において，東京大学大学院 江崎浩 教授，平木敬 教授，石川正俊 教授，稲葉雅幸 教授には多数の貴重な御教示と御助言を賜りました．ここに深謝の意を表します．

本研究を遂行する上で，数々の有益な御助言を頂いた University College London の Anthony Finkelstein 教授，The Open University の Bashar Nuseibeh 教授，国立情報学研究所 石川冬樹 准教授，同研究所 鄭顕志 助教，医薬基盤研究所 Johan Nyström-Persson 博士，グーグル株式会社 Adrian Klein 博士，久保淳人 博士に心から感謝の意を表します．本位田研究室の前澤悠太 氏，清水遼 氏，小林努 氏とは，ソフトウェア工学グループの研究活動を通じて，多くの有意義な議論をさせて頂きました．心から感謝いたします．その他，研究室セミナーにて貴重なコメントを頂いた先生方と学生の皆様，そして研究生活を送る上でお世話になった秘書の方々に心から感謝いたします．

本論文は，筆者がキヤノン株式会社に勤務しながら，社会人学生として研究を行った成果をまとめたものであり，関連部門の多くの方から御助力と御指導を賜りました．会社の業務を続けながら博士課程で学ぶことを快く許して下さり，大学での研究を助成して下さいました谷泰弘 取締役，岩渕洋一 執行役員に深謝の意を表します．ソフトウェア基盤第 6 開発部 海老原一之 部長には，企業研究者としての御自身の経験に基づいて多数の有益な御助言をいただきました．深謝いたします．ソフトウェア基盤第 1 開発部 佐々木誠司 部長，ソフトウェア基盤 61 開発室 高山正 室長，石井克己 氏（Canon U.S.A., Inc. Principal Scientist，元キヤノン株式会社 ソフトウェア基盤 11 開発室長）には，会社の業務と研究生活の両立を図るために様々な面で御協力を賜りました．心から感謝いたします．そして，本研究を進める上で日頃から御協力いただいたデジタルシステム開発本部の先輩，同僚諸氏に心から感謝いたします．

最後に，長い期間に渡って研究生活を見守り支えてくれた妻 紀子，息子 哲，娘 文乃に心から感謝します．皆さんの協力なしに，本論文を完成させることはできませんでした．

目次

第1章

序論

1.1 研究課題

コンピュータの高性能化とともに，より複雑な問題がコンピュータ上で扱われるようになり，ソフトウェアの規模が増大の一途を辿っている．1968 年に NATO のソフトウェア工学会議において，この状況が “software crisis” として取り上げられて以来，信頼性の高いソフトウェアの効率的な開発を実現するために，ソフトウェア工学の分野で多くの研究が行われてきた．

ソフトウェアの再利用，すなわち “新たなソフトウェアを構築する際に既存のソフトウェア開発の生成物を利用すること”（Krueger [1]）は，ソフトウェア開発の生産性と品質を向上させるための重要なアプローチである．有効な再利用を行うためには，生成物の記述レベルを抽象化して，重要な観点に照準を合わせることが肝要である．抽象化によって，着目する観点以外の詳細を意識せずに，生成物を再利用することができる．例えば，高水準のプログラミング言語を用いてソフトウェアを実装する場合には，機械語やアセンブリ言語を利用する際に必要となる，ハードウェアに依存した低水準の操作を意識せずに，対象システムの本質的なロジックの記述に注力できるため，より高い生産性と品質を達成することが可能である．

再利用のために用いられる技術は，パターンに基づく生成と，構成要素の合成の 2 つの技術に大別される（Biggerstaff et al. [2]）．前者は，テンプレートや変換ルールを用いて，ソフトウェアの自動生成を行う技術である．一方，後者は，ソフトウェアのブロックを可能な限り変更せずに組み合わせて，大規模なソフトウェアを構築する技術である．合成による再利用を系統的に行う方法として，ソフトウェアの構成要素を部品（コンポーネント）として開発し，コンポーネントを合成してソフトウェアシステムを構築する，コンポーネントベース開発技術の研究が行われてきた．コンポーネントベース開発では，抽象化されたコンポーネント仕様に基づいて再利用が行われ，仕様の実現方法はコンポーネント利用者から隠蔽される．ビジネス分野における業務の情報処理を担う情報システムの開発では，対象領域の業務の実体やプロセスなどの概念がビジネスコンポーネントとして仕様化され，ビジネスコンポーネントを合成してシステムの構築が行われる（Herzum et al. [3]）．

近年，モデル中心の開発アプローチであるモデル駆動工学（MDE: Model-Driven Engineer-

ing）の発展とともに，高抽象度のモデリング言語を用いたコンポーネント仕様の記述方法が確立され，大規模な分散システムの開発における，異なるプログラミング言語や動作プラットフォームに跨ったコンポーネントの再利用が可能になった（Schmidt [4]）．コンポーネントの性質は多面的であり，それぞれの側面ごとに適切なモデリング方法が必要である．OMG が策定した統一モデリング言語（UML: Unified Modeling Language）は，種々の用途に応じたグラフィカルな記法を備えている．また，制約記述言語 OCL（Object Constraint Language）を用いて，UML モデルの制約を宣言的に記述することができる．情報システムを構成するビジネスコンポーネントの仕様記述には，UML/OCL が広く用いられている．

コンポーネントベース開発では，仕様モデルに基づいてコンポーネントの実装・利用を行うため，仕様定義段階で，1) 個々の仕様モデルの表現内容について，モデルの記述者の意図と利用者の解釈が一致していること，2) 合成されたコンポーネントの仕様モデルが，システムに要求される性質を満たすこと，の 2 つの整合性を確保することが，ソフトウェアの品質を高める上で重要である．1) の解釈の一致は，2) の合成モデルの整合性を議論する上での前提となる．

情報システムを構成する，業務データの管理を責務とするビジネスコンポーネントの振舞いモデルは，OCL を用いて記述され，管理データに対する代入・取得・照合・削除のアクセスを表現する．従って，OCL 記述のデータアクセスの表現に関して，記述者の意図と利用者の解釈が一致することが必要不可欠である．また，システムの横断的な性質である，コンポーネント間に跨るデータの参照整合性は，ビジネスロジックの実行を責務とするビジネスコンポーネントの振舞いの中で，適切な手順でデータアクセスを行うことによって実現される．従って，UML Activity を用いて記述される，それらの振舞いの組合せについて，データアクセス操作の呼び出し手順の正しさを確認する必要がある．

宣言的な仕様記述言語には，手続きの情報を排除した汎用的な形式で，コンポーネントの振舞いの仕様を記述できる利点があるが，振舞いを一意に特定できない “underspecied” な記述を受容するため，モデル要素の状態が確定しない可能性がある（Wieringa [5]）．そのため，1) のモデル記述者の意図と利用者の解釈が一致せず，不整合が生じる恐れがある．また，大規模なシステムの開発では，多数のコンポーネントの仕様モデルが合成される．人手によって，全ての仕様モデルの内容を詳細に把握して，2) のシステム要求の充足を正確かつ網羅的に確認することは困難な作業である．仕様定義段階で 1)，2) の不整合を除去しきれず，欠陥を含んだ仕様モデルが次工程に引き渡されると，後に仕様モデルの再定義を行うための工程の手戻りが発生して，開発生産性の低下を招いてしまう．

本博士論文では，仕様定義段階でビジネスコンポーネントに跨る参照整合性を確立して，情報システムのコンポーネントベース開発の生産性と品質の向上を実現するために，次の 2 つの整合性の判定方法の研究に取り組む．

- OCL で記述されたコンポーネントの振舞いモデルの，データアクセスに関する記述者の意図と利用者の解釈の整合性
- UML Activity で記述されたコンポーネントの振舞いモデルの組み合わせにおける，データアクセス操作の利用手順の整合性

本論文では，第 1 の整合性の判定を支援する OCL 記述の解析手法と，第 2 の判定を支援する UML Activity の解析手法を提案する．これらの解析手法は，コンポーネントの仕様モデルを解析して，整合性の判定を容易に行うための情報を提供する．2 つの手法はいずれも静的な解析方法であり，仕様定義段階でソフトウェアを動作させずに整合性の判定を行うことができる．

本研究では，解析の実行を自動化する支援ツールの開発を行った．ツールを自動実行して，その出力情報を用いて整合性の判定を行うことにより，判定の正確性と効率の向上が可能になる．さらに，本論文では，実システムの開発に対して実施した評価実験の結果を報告し，実開発における手法の有用性を確認する．

1.2 本研究の貢献

本研究で提案する UML/OCL の解析手法の，従来研究に対する新たな貢献について述べる．

OCL 記述の解析

本提案手法は，抽象解釈技術を用いて，OCL 記述の意味の近似的な解釈を行う．OCL 記述に対する抽象解釈の適用は，従来の研究と異なる，新しいアプローチである．さらに，提案手法は，OCL 抽象構文木上のノード値の由来とデータの流向を特定する方法を，新たに導入する．本手法のアプローチによって，OCL 言語仕様のある範囲において，次の 4 つの要件が実現される．特に，実用的な計算時間内での完全な自動解析を実現するアプローチは，これまで存在しなかった．

- 取りうるオブジェクト状態に対する完全な解析
- 利用者の介入を必要としない，実用的な計算時間での自動解析
- OCL の言語特性である文脈依存性・非決定性・不完全性の厳密な取扱い
- 代入，照合，取得のアクセス種別の識別

Activity の解析

本提案手法は，参照整合性を破壊しうるデータアクセスと，破壊を防ぐために必要なデータアクセスの組を定義し，Activity の属性依存のデータフロー解析の方法を導入する．これらは，UML Activity の組合せを対象として，関係データベースの参照整合性の実現方法と，データフロー解析の技術を応用したものであり，従来にない新しい応用の形態である．さらに，提案手法は，データアクセス操作の引数の到達定義と，操作呼び出しの履歴情報を利用する，データアクセス手順の整合性の判定方法を新たに導入する．本手法のアプローチによって，次の 3 つの要件が実現される．

- ビジネスコンポーネントに跨ったデータアクセス順序の整合性の判定
- Activity 中のオブジェクトの属性値の伝搬経路の把握
- Activity 中のデータの流れに依存するデータアクセス手順の整合性の判定

1.3　論文の構成

本論文の構成と，各章の概要を以下に示す．

第 2 章：**背景・関連研究**

研究の背景と関連研究の説明を行う．情報システムのコンポーネントベース開発の特徴を述べた後，UML/OCL に基づくビジネスコンポーネントの仕様モデルの定義を行い，例題モデルを導入する．コンポーネント間の参照整合性を確保するために必要な振舞いの整合性について論じる．そして，例題モデルを用いて，振舞いの整合性判定の難しさを説明する．関連研究として，従来の整合性判定の技術と，本研究の土台となる抽象解釈とデータフロー解析の説明を行う．

第 3 章：**OCL 記述の解析**

OCL 記述の解析手法を導入する．解析の入出力，技術課題，課題解決のためのアプローチを述べ，手法の詳細を説明する．提案手法は，コンポーネント操作の事後条件の OCL 記述を静的に解析して，代入・取得・照合のデータアクセスを識別する．抽象解釈技術を用いて，OCL 記述が表現する制約を満たすオブジェクト状態を，抽象領域上で網羅的に扱う解析方法を導入する．また，OCL 抽象構文木上のノード値の由来とデータの流向を特定する方法を定義する．手法の支援ツールについて述べる．

第 4 章：**Activity の解析**

Activity の解析手法を導入する．解析の入出力，技術課題，課題解決のためのアプローチを述べ，手法の詳細を説明する．提案手法は，コンポーネントの振舞いの Activity の組合せを静的に解析して，合成されたモデルの中で起こりうるデータアクセス操作の利用手順の正しさを判定する．参照整合性を破壊しうるデータアクセスと，破壊を防ぐために必要なデータアクセスの組を定義する．また，Activity の属性依存のデータフロー解析の方法を導入し，データアクセス操作の引数の到達定義と，操作呼び出しの履歴情報を求めて，整合性判定を行う方法を定義する．手法の支援ツールについて述べる．

第 5 章：**評価**

OCL 記述の解析手法と Activity の解析手法の評価を行う．2 つの解析手法の妥当性，能力，適用範囲，利用方法を論じる．さらに，実システムのコンポーネントベース開発において，評価実験を行った．実験の内容と結果を説明し，手法の有用性について考察を行う．最後に，関連研究について述べる．

第 6 章：**結論**

本研究の結果をまとめ，今後の課題と展望を述べる．

第2章

背景・関連研究

本章では，研究の背景と関連研究を説明する．2.1 節で，ソフトウェアの再利用に基づく開発方法であるコンポーネントベース開発の説明を行う．コンポーネント，コンポーネントモデル，インタフェース，契約，関心の分離，階層アーキテクチャ，開発プロセスについて論じる．2.2 節で，情報システムのコンポーネントベース開発の特徴を説明する．情報システムの定義，N 層アーキテクチャ，システムを構成するビジネスコンポーネントについて述べる．2.3 節で，UML/OCL に基づくビジネスコンポーネントの仕様モデルを定義して，例題モデルを導入する．永続データへのアクセスとビジネスロジックの振舞い仕様は，OCL の事後条件と UML Activity 図を用いて，それぞれ記述される．2.4 節で，本論文の主題である，コンポーネント間の参照整合性を確保するための振舞いの整合性について論じる．例題モデルを用いて振舞いの整合性判定の難しさを説明し，仕様モデルの不整合が引き起こす開発の手戻りについて述べる．2.5 節では，関連研究について論じる．OCL モデルの誤り防止と，*Activity* の処理順序の検証に関する従来の研究を述べた後に，本研究の土台となる抽象解釈とデータフロー解析について説明する．そして，関係データベースの表間の参照整合性を実現するための仕組みである参照制約動作の概念を説明する．

2.1　コンポーネントベース開発

コンポーネントベース開発（CBD: Component based development）は，ソフトウェア部品であるコンポーネントを組み合わせて，ソフトウェアシステムを構築する開発方法である (Heineman et al. [6]，Sametinger [7])．1968 年に NATO のソフトウェア工学会議において，McIlroy [8] がその概念を提唱して以来，CBD はソフトウェア工学の一分野として位置付けられ，様々な研究が行われてきた．CBD は，機械部品や電子部品を組み合わせて，大規模で複雑な機器を作り上げるハードウェア開発のアナロジーであり，システムを一から個別開発する代わりに，コンポーネントを再利用してシステムを構築することにより，開発の生産性とソフトウェアの品質を向上させることを狙いとしている．

これまでに，組み込み機器ソフトウェア，通信・医療・生産のシステムソフトウェア，企業情報システムなど，多様な産業用ソフトウェアの開発に CBD が適用され，数多くのケースで，

開発期間の短縮，開発コストとメンテナンスコストの低減，ソフトウェアバグ密度の減少が確認されており（Griss [9]，Agresti et al. [10]，Frakes et al. [11]，Browne et al. [12]），その有用性が広く認知されている．

コンポーネント

“コンポーネント” は，ソフトウェアの部分を指すために頻繁に用いられる用語であるが，その正確な意味については，合意された定義は未だ存在しない（Sametinger [7]，Vitharana et al. [13]，Broy et al. [14]）．実際に，以下のような複数の異なる意味が，文脈に応じて使い分けられている．

- ソースコードの断片や，クラス，モジュール等のプログラムの構成単位
- グラフィカル・ユーザ・インタフェースのボタン等のオブジェクト
- 商用のソフトウェア部品（COTS: Component Off The Shelf）

一方で，ソフトウェアの再利用を主題とする CBD の文脈に限れば，“コンポーネント” はアーキテクチャを構成する再利用ユニットの意味で用いられる．CBD の分野の中でも，コンポーネントの定義には多数のバリエーションが存在するが，一般にそれらの間の差異は小さく，概ね共通の概念が共有されている．ここでは代表的な Bachmann et al. [15] の定義に基づいて，CBD におけるコンポーネントの概念を説明する．Bachmann らはコンポーネントを，実装とアーキテクチャ上の抽象概念の 2 つの視点を組み合わせたものとして位置付け，その性質を次のように規定した．

- 特定の機能を実装し，その内部構造や機能の実現手段を隠蔽する．
- 公開された仕様に基づいて，第三者がコンポーネントを合成[*1] することが出来る．
- 特定のコンポーネントモデルに準拠する．

第 1，第 2 の性質によって，コンポーネントの仕様と実装が明確に分離され，コンポーネント内部の知識無しにブラックボックスの部品として利用することが可能になる．そのような利用は，大規模なソフトウェアシステムを効率的に開発するために不可欠のものである．コンポーネントの仕様は，後述のインタフェースとして表現される．第 3 の性質におけるコンポーネントモデルは，コンポーネントの意味論と構文を定義し，コンポーネントの合成，相互通信，配置の標準的な方法を定める．コンポーネントモデルに従って，コンポーネントを系統的に開発し，組み合わせることが可能になる．コンポーネントモデルのメカニズムの実装は，コンポーネントフレームワーク，又はコンポーネントプラットフォームと呼ばれる（Bachmann et al. [15]，Sametinger [7]）．

コンポーネントモデルの区分

一般的なコンポーネントモデルでは，コンポーネントは機能をサービスとして提供し，同

[*1] CBD 関連の文献では，部品の組立て（assembly）を表す用語として，主に “合成（composition）” と “統合（integration）” が用いられる．本論文では，これらの用語を適宜使い分けるが，両者に実質的な違いは無い．

時に，サービスの実現に必要な他のコンポーネントのサービスを要求する．提供サービスと要求サービスの依存関係は，コンポーネントの仕様としてインタフェースに定義される．Lau et al. [16] はコンポーネントの意味論の観点から，コンポーネントモデルを次の 3 つに区分した：

1) コンポーネントが，オブジェクト指向プログラミング言語のクラスである区分，
2) コンポーネントが，オブジェクトのように振る舞う，実行時の実体である区分，
3) コンポーネントが，計算，制御，データ管理等のアーキテクチャの構成単位である区分．

区分 1) では，コンポーネントの仕様と実装を，単一のプログラミング言語を用いて記述する．EJB コンポーネントを Java 言語のクラスとして定義，実装する EJB: Enterprise JavaBeans（Monson-Haefel [17]）は，この区分に含まれる．

区分 2) では，実装に中立なインタフェース記述言語（IDL: Interface Definition Language）を用いてコンポーネントの仕様を定義し，各種プログラミング言語を用いて実装する．代表例として，IDL に WSDL: Web Service Description Language を用いる WebService（Curbera et al. [18]）や，OMG IDL を用いる CCM: CORBA Component Model（OMG [19]）が挙げられる．

区分 3) では，アーキテクチャ記述言語，すなわち ADL: Architecture Description Language（Clements [20]，Medvidovic et al. [21]）を用いて，インタフェースと，コンポーネント間の接続を形式的に定義し，実装に用いるプログラミング言語や実装方法には言及しない． 本研究では，区分 3) の中で，統一モデリング言語 UML: Unified Modeling Language（OMG [22]）を ADL として用いるコンポーネントモデルを扱う．UML は，産業界で広く用いられている標準的なオブジェクトモデリング言語である．

インタフェース

インタフェースは，コンポーネントの仕様を表現する．各コンポーネントは 1 以上の複数のインタフェースを実装し，コンポーネントの利用者は，インタフェースで定義された仕様に基づいてコンポーネントを合成する．前述した通り，コンポーネントは仕様と実装の二つの意味を併せ持つ概念であるが，“コンポーネント C はインタフェース I を実装する”というように，実装の側面を明示的に強調している場合は，“コンポーネント”は実装上の再利用ユニットを意味する．

Beugnard et al. [23] は，インタフェースが備える 4 種類の特性を定義した：

1) コンポーネントの操作[*2]のシグネチャ，
2) コンポーネントの操作の振る舞い，
3) コンポーネントの操作の間の同期，並行性，
4) コンポーネントの操作の品質特性．

[*2] サービスの提供形態を操作と呼ぶ．区分 1) のコンポーネントモデルではクラスのメソッドが，区分 2) の WebService や区分 3) の UML ベースのコンポーネントモデルでは *Operation* が，操作に対応する．

特性 1) のシグネチャは，操作の識別子，入出力パラメータ，戻り値，データ型，例外の情報であり，プログラミング言語や IDL を用いて API: Application Programming Interface として定義される．特性 1) が操作の構文を定義するのに対して，特性 2)，3)，4) は構文には表れない，操作の意味を定義する．

特性 2) は，操作の実行がもたらす結果を規定する．Eiffel（Meyer [24]）などの一部のプログラミング言語では，特性 2) を操作毎の事前・事後条件として定義することが出来る．しかし大多数のプログラミング言語や IDL は，特性 2) を表現する能力を備えていない．Java 言語に関しては，特性 2) を記述するための言語拡張として JML: Java Modeling Language（Leavens et al. [25]）や iContract（Kramer [26]）が提案されている．区分 3) の UML ベースのコンポーネントモデルでは，OCL: Object Constraint Language（OMG [27]）を利用することが出来る．OCL は UML モデルの制約を表現するための宣言的な仕様記述言語であり，IBM の Syntropy method（Cook et al. [28]）に起源を持つ．その他に，形式仕様記述言語である VDM++，Z を用いて，特性 2) を記述する方法が提案されている（Goldsack et al. [29], Johnson et al. [30]）．

特性 3) は，単一のコンポーネントまたは複数のコンポーネントの操作の呼び出しの間の並行性の情報であり，区分 3) のコンポーネントモデルについて，プロセス代数の CSP（Communicating Sequential Processes），π 計算を用いた記述が提案されている（Allen et al. [31]，Canal et al. [32]，Henderson et al. [33]，Lumpe et al. [34]）．

特性 4) は，操作の性能，再利用性，可用性，信頼性，セキュリティ，使用性，出力値の精度などの品質を規定する．Choukair et al. [35] は，区分 2) の CORBA コンポーネントの操作の性能を表現するための実時間拡張を提案した．Franch [36] は区分 3) のコンポーネントモデルについて，操作の性能や信頼性の形式的な記述方法を提案した．

契約

インタフェースは，サービスを提供するコンポーネントだけではなく，サービス利用側のコンポーネントの責務も規定する（Beugnard et al. [23]）．Holland et al. [37, 38] は，この相互責務の考え方を契約と名付けた．サービス利用側のコンポーネントは，操作を呼び出す時点で，特性 2) で定義される操作の事前条件を満たす必要がある．一方，サービス提供側のコンポーネントは，操作の実行後に特性 2) の事後条件を満たす義務を負う．また双方のコンポーネントは，特性 1) の操作のシグネチャの構文に従わなければならない．契約は通常，個々のコンポーネント毎または提供サービス毎に定義される（Meyer [24]）．しかし Holland et al. [37, 38] の提案のように，コンポーネント間の相互作用について契約を定義する方法も存在する．

関心の分離の原則

Dijkstra [39] は，プログラムを開発する上で様々な関心を分離して扱う，関心の分離（SoC: Separation of Concerns）の原則を提唱した．この原則は，規模の大きな問題を，独立した小さな問題の組み合わせに分割して解決を図る，分割統治の考え方に基づいている．CBD では

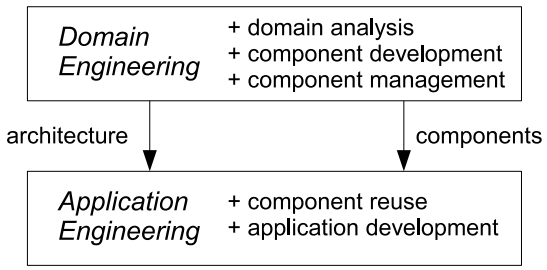


図 2.1. CBD の開発プロセス

通常，関心の分離の原則に従って，個々のコンポーネントが扱う問題を分離して，コンポーネントの独立性を高めることが重要視される（Heineman et al. [6]）[*3].

階層アーキテクチャ

Garlan et al. [41, 42] は，個々のコンポーネントが想定するコンポーネントモデルとアーキテクチャの違いによって，コンポーネントの合成が困難になる “Architectural mismatch” の問題を指摘した．コンポーネントを効果的に合成するためには，個々のコンポーネントの関心の分離に加えて，システムのアーキテクチャが重要な役割を持つ．特に，Batory et al. [43] が提案した階層アーキテクチャは，各コンポーネントが同一あるいは下位の階層に位置するコンポーネントにのみ依存する DAG: Directed Acyclic Graph の構造を持ち，コンポーネント間の干渉が限定される．この特徴によって，コンポーネントの合成の難しさを軽減することが出来るため，一般的な CBD では階層アーキテクチャが採用されている．

開発プロセス

コンポーネントの系統的な再利用を行うためには，再利用資産の開発と管理を行うドメインエンジニアリングと，再利用資産を利用して個々のシステムを構築するアプリケーションエンジニアリングの 2 つの活動[*4] が必要である（Sametinger [7]）．両者の関係を図 2.1 に示す．

ドメインエンジニアリングでは，対象ドメイン[*5] の分析を行い，ドメインの共通性と可変性を抽出する．そしてアーキテクチャを決定すると共に，コンポーネントを識別し，必要なコンポーネントの開発（仕様定義・設計・実装・テスト）と COTS の調達を行う．さらにコンポーネントの管理，すなわち品質保証，保管，分類，カタログ化，版管理を行う．一方，アプリケーションエンジニアリングでは，システムの要件に合わせて適切なコンポーネントを選択，再利用してシステムを構築する．実際にはコンポーネントを無変更で再利用できるケースは少なく，アダプテーション（Bosch [44]）やラッピング（Spitznagel et al. [45]）を施した上

[*3] CBD の他に，関心の分離の原則に基づくソフトウェア工学の技術として，ソフトウェアの部分に跨る横断的な関心を解決するアスペクト指向技術（Kiczales et al. [40]）が知られている．

[*4] これらの活動は，文献によって異なる名前で言及されることがある．例えば Heineman et al. [6] では前者の活動を Component Process，後者の活動を Solution Process と呼んでいる．

[*5] 給与計算システムや生産管理システム等のシステムの種別を表すドメイン（vertical domain）と，数値計算ライブラリやデータベースソフトウェア等の汎用ソフトウェアの種別を表すドメイン（horizontal domain）が存在する（Sametinger [7]）．

で再利用することが多い．

CBD では，コンポーネントの仕様を定義した後に，ドメインエンジニアリングにおける各コンポーネントの実装と，アプリケーションエンジニアリングにおけるシステムの構築を，独立に実施することができる．開発の早期にコンポーネントの単位で作業を分担することによって，大規模なシステム開発を効率的に進めることが可能である．

2.2　情報システムのコンポーネントベース開発

情報システム（IS: Information System）とは，データの入力を受付けて，通信ネットワークで接続された複数のコンピュータを用いて，データの変換・蓄積・出力の情報処理を行うシステムである．広義には全てのコンピュータシステムが情報システムであるが，狭義には，企業などの組織が所有する大規模なコンピュータシステムを情報システムと呼ぶ[*6]．本節では，狭義の情報システムの CBD について説明する．

情報システムは，機能と運用方法の違いによって，次の 2 つに区別して扱われることがある．

- 基幹系システム：組織の主業務の情報処理を担うシステム．典型的な例として，銀行業の預貯金システムや，運輸業の運行管理システムなどが挙げられる．
- 情報系システム：主業務に付随する業務の情報処理を担うシステム．人事管理システム，購買管理システムなどが挙げられる．

従来，高い信頼性と性能の安定性が要求される基幹系システムは，ベンダー独自のハードウェアやオペレーティングシステム（OS: Operating System）を備えるメインフレームを用いて構築されてきた．しかし 1990 年代以降，オープン標準に準拠したハードウェアや OS を用いてオープンシステムとして構築されるケースが増えており，技術面で情報系システムとの差異が小さくなりつつある．また基幹系システムのデータを情報系システムで活用するなど，2 つのシステムの融合が進行している．本研究では両者の違いについて言及せず，情報システムとして一律に扱う．

N 層アーキテクチャ

情報システムは通常，複数の層（tier）を有する N 層アーキテクチャに基づいて構築される．N 個の層は，関心の分離の原則に従って，それぞれ異なる責務を持ち，システムを構成するコンポーネントは何れかの層に属する．一般的に，システムのスケーラビリティを確保するために，各層のコンポーネントは複数のコンピュータに分散配置され，コンポーネント間の通信はネットワークプロトコルや分散ミドルウェアを介して行われる．代表的な 4 層アーキテクチャの構造を図 2.2 に示す．各層は以下の役割を持つ．

- User Interface 層 : Web ブラウザや専用端末画面を通して，ユーザとの相互作用を担う．
- Presentation 層 : ユーザの入力データを Business Logic 層に伝達し，Business Logic

[*6] 狭義の情報システムは，エンタープライズシステムとも呼ばれる．

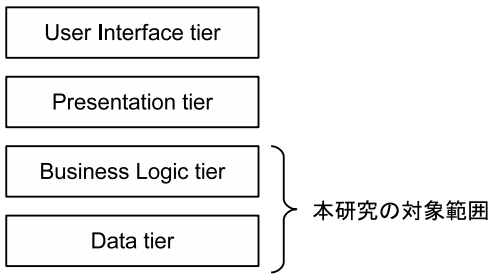


図 2.2. 4 層アーキテクチャ

表 2.1. 層（tier）の名称

本論文	Emmerich [46]	Herzum et al. [3]	Cheesman et al. [47]
User Interface 層	Display 層	User 層	–
Presentation 層	–	Workspace 層	User Dialog 層
Business Logic 層	Business Object 層	Enterprise 層	System Service 層
Data 層	Persistence 層	Resource 層	Business Service 層

層の処理結果から出力データを生成する．

- Business Logic 層：Data 層が管理するデータにアクセスし，ビジネスロジック，すなわちシステム固有の処理フローやルールを実行する．
- Data 層：システム内のデータの永続化，管理を行う．

4 層アーキテクチャの定義には，文献によって幾つかのバリエーションが存在する．各バリエーションの間では，層の名称が一部異なるが，層毎の役割は概ね一致している．バリエーション間の層の名称の対応を表 2.1 に示す．

User Interface 層と Presentation 層は，個々のユーザの要求を個別に扱う．一方，Business Logic 層と Data 層は，システム全体のデータの整合を保ちながら，複数の要求を同時並行に扱う．複雑な処理を行う Business Logic 層と Data 層は，情報システムの中核であり，本論文では，この 2 層に属するコンポーネントを対象として研究を行う．なお以降では図 2.2 の 4 層アーキテクチャを用いて説明を行うが，Business Logic 層と Data 層は，N の値に依らず，あらゆる情報システムで用いられる．従って，本研究の成果は N 層アーキテクチャの情報システム一般に適用可能である．

ビジネスコンポーネント

情報システムの CBD において，対象領域の業務の実体やプロセスなどの概念を表現・実装するコンポーネントは，ビジネスコンポーネントと呼ばれる（Herzum et al. [3]，Albani et al. [48]，Wang et al. [49]，Heineman et al. [6]）．ビジネスコンポーネントは，その機能から，次の 2 種類に分類される．

表 2.2. ビジネスコンポーネントの名称

本論文	Wang et al. [49]	Herzum et al. [3]	Cheesman et al. [47]
Process component	–	Process business component	System component
Data component	Entity component	Entity business component	Business component

- DC (Data component) : Data 層に位置する，永続データの管理を行うコンポーネント．
- PC（Process component）: Business Logic 層に位置する，ビジネスロジックを実行するコンポーネント．

文献によって，ビジネスコンポーネントの定義に幾つかのバリエーションが存在する．各バリエーションの間で，分類の名称が一部異なるが，分類の内容は一致している．バリエーション間の名称の対応を表 2.2 に示す．

ビジネスコンポーネントは，2.1 節の区分 3) のコンポーネントモデルにおける，アーキテクチャの論理的な構成単位である．情報システムの CBD では，問題領域をビジネスコンポーネントに分割し，ビジネスコンポーネントを統合することによってシステムを構築する．そのような CBD を行うための方法論として，UML Components（Cheesman et al. [47]），Business Component Factory（Herzum et al. [3]），Catalysis（D'souza et al. [50]）が提案されている．

再利用可能なコンポーネントを識別することは，CBD の中心的な作業である．Wang et al. [49] は，コンポーネントの識別方法を次の 2 通りに分類している．

- FI（forward identification）: ソフトウェアをスクラッチ開発する場合に，システムの要求モデルに基づいてコンポーネントを新規に識別する．
- RI（reverse identification）: 既存のソフトウェア資産から，リーバスエンジニアリング等によりコンポーネントを識別する．

UML Components は，FI に特化した CBD 方法論である．一方 Catalysis は，FI を行う手順として “Build route” を，RI を行う手順として “Assemble route” を定めている．Wang et al. [49] では，FI の代表的な手法を次の 3 通りに分類している：

- FODA（Kang et al. [51]）等のドメイン分析に基づく手法，
- 凝集度・結合度ベースのクラスタリング分析に基づく手法，
- モデル要素間の CRUD 関係に基づく CRUD matrix 手法．

本研究では，ビジネスコンポーネントの仕様モデルの解析について論じるが，コンポーネントの識別方法に関する仮定を設けない．すなわち，FI または RI によって識別されたビジネスコンポーネントの仕様モデルを，区別することなく扱う．

2.3 ビジネスコンポーネントの仕様モデルと例題

UML/OCL に基づくビジネスコンポーネントの仕様モデルを定義して，例題モデルを導入する．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．

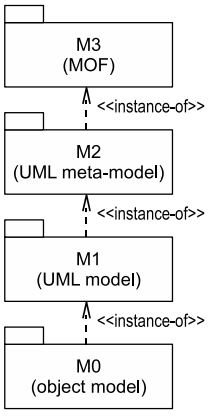
- 構造モデル：DC と PC の静的な構造を UML コンポーネント図として，DC の整合性制約を OCL 不変条件として，それぞれ記述する．また DC 間の参照をモデル化する．
- DC の振舞いモデル：DC の操作の振舞いを，OCL 事前・事後条件として記述する．
- PC の振舞いモデル：PC の操作の振舞いを，UML Activity 図として記述する．

本研究で扱う仕様モデルは，UML/OCL の記法と意味論に厳密に従っており，UML Components（Cheesman et al. [47]），Catalysis（D'souza et al. [50]）などの多くの情報システムの CBD 方法論のコンポーネントモデルに適合する．従って，これらの開発手法を採用する多数の実システムの開発で，本研究の提案手法を適用することが出来る．

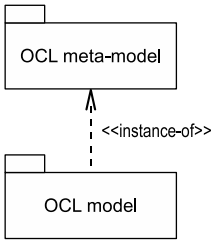
本節の構成は次の通りである．2.3.1 項で UML/OCL のメタモデルについて述べる．2.3.2–2.3.4 項において，構造モデル，DC の振舞いモデル，PC の振舞いモデルについて説明する．これらの項で導入する例題モデルは，本論文を通じて提案手法の説明を行う際に利用する．

2.3.1 UML/OCL のメタモデル

一般に，モデルの記法（Syntax）と意味論（Semantics）は，上位概念のメタモデルによって定義され，モデルとメタモデルの関係を規定するメタモデル階層が存在する．階層 N のモデルは隣接する上位階層 N+1 のインスタンスであり，同時に隣接する下位階層 N−1 のメタモデルである．



(a) UML のメタモデル階層（4 層）



(b) OCL のメタモデル階層

図 2.3. メタモデル階層構造

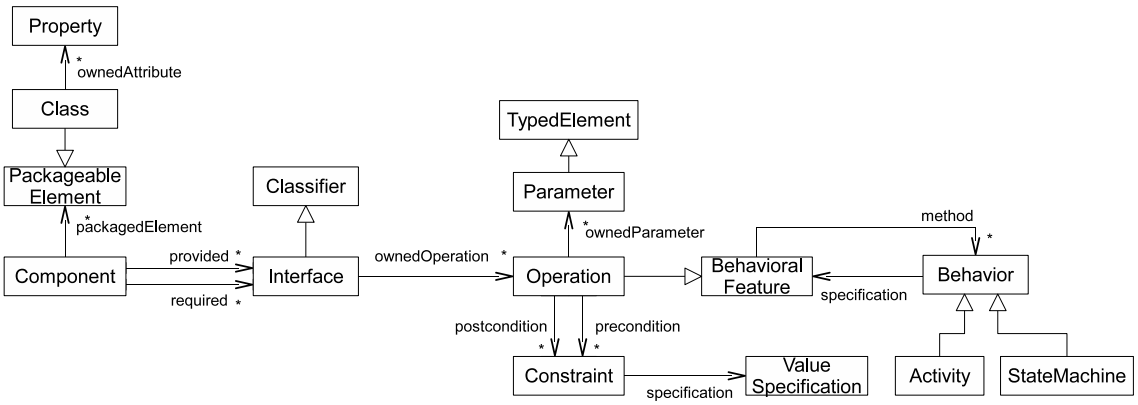


図 2.4. UML メタモデル

UML のメタモデル階層は，図 2.3(a) の 4 層構造を持つ（Object Management Group [52]）．最上位の MOF（Meta Object Facility）は，モデリングの核となる概念を定義する階層であり，MOF 自身のインスタンスである．すなわち MOF の定義は再帰的である．UML メタモデルは UML モデルの記法と意味論を定義する．UML モデルは，UML 利用者が記述する，対象システムの抽象表現である．オブジェクトモデルは UML モデルのインスタンスであり，システムの実行時に生成されるオブジェクト構成を表す．

図 2.4 に UML メタモデルの一部を示す[*7]．コンポーネント（*Component*）はインタフェース（*Interface*）を提供・要求し，*Interface* は複数の操作（*Operation*）を有する．各 *Operation* は複数のパラメータ（*Parameter*）を有し，その振舞いは事前・事後条件の制約（*Constraint*）あるいは *Activity*，*StateMachine* 等の *Behavior* として記述される．また *Component* は，その内部にクラス（*Class*）等の *PackageableElement* を有する．*Class* は属性として *Property* を有する．

OCL のメタモデル階層を図 2.3(b) に示す．OCL メタモデルは，OCL モデルの抽象構文と具象構文を定義する．OCL 言語仕様は MOF と UML の共通部分に基づいて定義されており，OCL 抽象構文メタモデルには UML メタモデルの要素が取り込まれている．OCL モデルは，MOF/UML メタモデル/UML モデルの階層に対する制約を，不変条件，操作の事前・事後条件として記述する．

図 2.5 に，OCL 抽象構文メタモデルを示す．抽象構文は，OCL 表現（*OclExpression*）の組合せによって構成される．図中のグレー部分は UML メタモデルから取り込まれた要素である．*OperationCallExp* には，オブジェクトの集まり（*Collection*）に対する全称量化演算 `forAll` と存在量化演算 `exists` が含まれる．つまり，OCL は一階述語論理の記述能力を備えている．

OCL モデルは，構文解析を経て抽象構文木として表現される．抽象構文木のノード（OCL

[*7] 本論文では，UML/OCL メタモデルの要素を本文中で参照する際にイタリック体を，UML/OCL モデルの要素を参照する際にタイプライタ体を，それぞれ使用する．

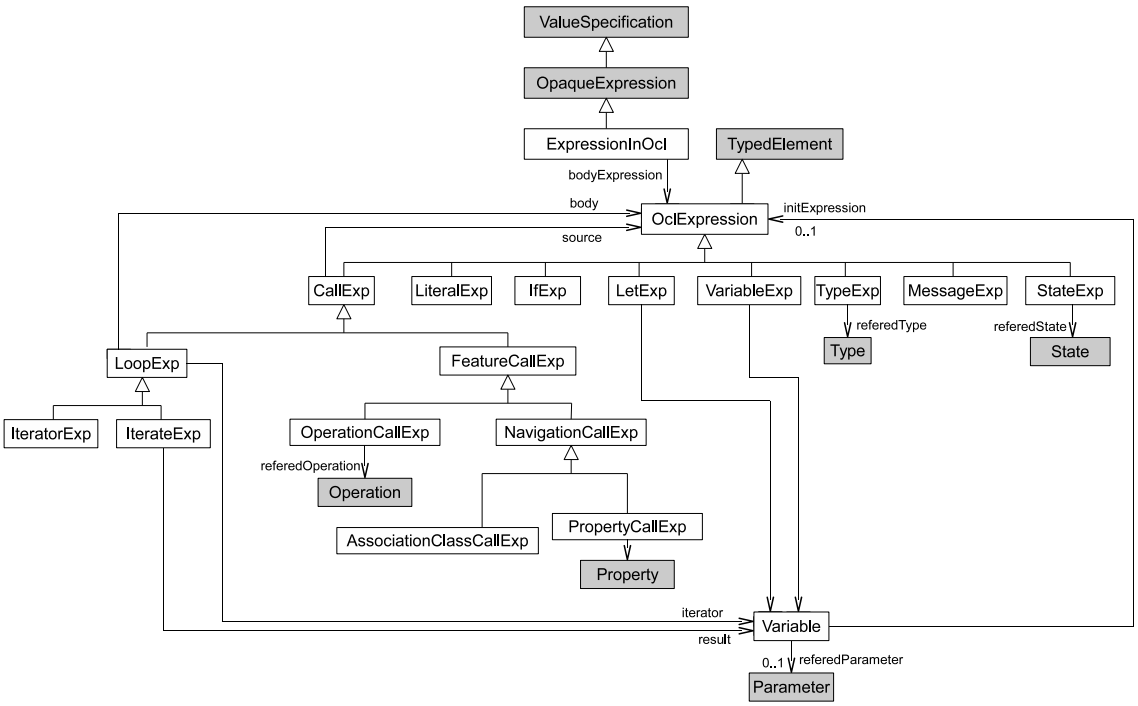


図 2.5. OCL 表現の抽象構文メタモデル

表現ノード）は，OCL メタモデルで定められている OCL 表現のインスタンスであり，各 OCL 表現ノードは単一の型を持つ．OCL は宣言的な仕様記述言語であり，OCL 表現ノードを直接実行することはできない[*8]．制約の対象モデルのインスタンスに対して抽象構文木を評価することによって，各 OCL 表現ノードの値が決定される．全ての OCL 表現ノードは副作用を持たず，対象モデルのインスタンスは評価を通じて変更を受けない．

OCL 表現ノードの値と型の対応を表 2.3 に示す．*PrimitiveType* は UML の基本型（*Boolean, Integer, Real, String, Enumeration*）を表す．*CollectionType* はオブジェクトの集まりを示す抽象型であり，その具象型として，集合（*Set*），順序集合（*OrderedSet*），多重集

表 2.3. OCL 表現の値と型

値	型
PrimitiveValue	*PrimitiveType*
TupleValue	*DataType*
ObjectValue	*Class*
CollectionValue *(SetTypeValue,BagTypeValue, SequenceTypeValue)*	*CollectionType* *(Set,OrderedSet,Bag,Sequence)*
OclMessageValue	*OclMessageType*
OclVoidValue	All types

[*8] OCL に命令型の拡張を行った言語として，モデル間の変換を行うための標準言語 QVT: Queries/Views/-Transformations（OMG [53]）がある．

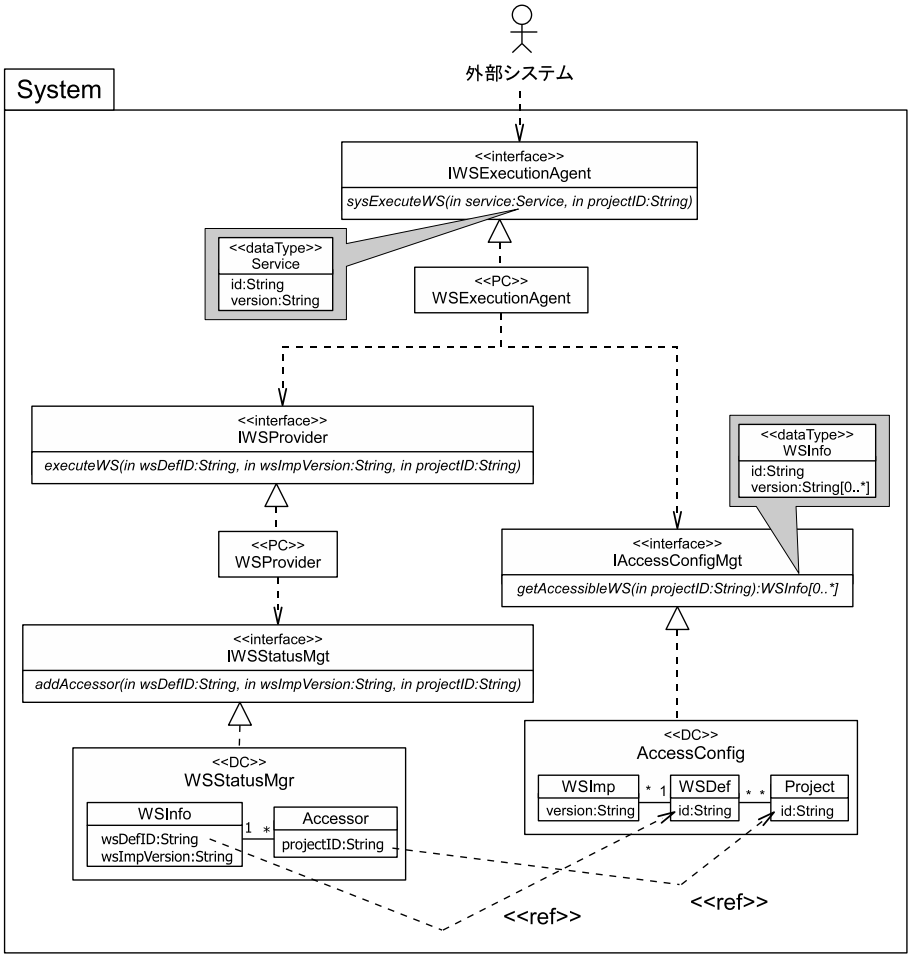


図 2.6. コンポーネント図

合（*Bag*），列（*Sequence*）の 4 種類が存在する．OCL の論理体系は 3 値論理であり，*Boolean* ノードが真・偽・*OclVoidValue* をとる．*OclVoidValue* は未定義値，あるいは 0 除算等により発生する例外を表す値であり，全ての型について定義される．

2.3.2　構造モデル

構造モデルは，DC のデータモデルと，DC・PC が提供・使用するインタフェースを UML コンポーネント図で表現する．永続データの管理を行わない PC には，インタフェース記述のみ定義される．図 2.6 に構造モデルの例を示す．インタフェース，DC，PC はそれぞれ，ステレオタイプ ⟨⟨`interface`⟩⟩，⟨⟨`DC`⟩⟩，⟨⟨`PC`⟩⟩ によって区別される．

データモデル

データモデルは DC が扱うデータを *Class* として，データの関係を順序無しの関連として表現する．各 *Class* は属性定義を持つが，操作を持たない．データへのアクセスは，DC の操作を通じて行われる．各 *Class* は別個のデータを表すために定義され，*Class* 間には継承関係は

存在しない.

各 DC のデータモデルは独立しており, データモデル間に直接の関連は存在しない. Herzum et al. [3] は, このような DC ごとの分散データ管理に基づくアーキテクチャ上のアプローチを *Islands of Data* と呼んでいる. 一方, 対極的なアプローチとして, システム全体で単一のデータモデルを用いて統合データ管理を行う方法がある. 分散管理方式は統合管理方式に比べて, 各 DC のデータモデルを独立に維持管理することができる. 従って, 大規模システムの CBD において開発の作業分担を行ったり, より細かい粒度でデータモデルを再利用する上で, 大きな利点を持つ.

図 2.6 の例では, `AccessConfig` 内で `Project`, `WSDef`, `WSImp` のデータオブジェクトが, `WSStatusMgr` 内で `WSInfo`, `Accessor` のデータオブジェクトが, それぞれ分散管理される. `AccessConfig` は, プロジェクト毎に利用可能な WebService の情報を保持する DC である. `WSDef` と `WSImp` は, WebService の定義と実装を表すクラスであり, プロジェクトを表す `Project` との関連によって, 各プロジェクトが利用可能な WebService の情報を表現している. 一方, `WSStatusMgr` は, WebService の実行状態を管理する DC である. `WSInfo` と `Accessor` は, それぞれ実行中の WebService と実行依頼者を表すクラスである.

CBD では, コンポーネントの外部仕様として規定される性質と, 個々のコンポーネントの実装に依存する性質は, 明確に区別して扱われる. Bachmann et al. [15] は, 前者を “abstract interface”, 後者を “bound interface” と呼んでいる. DC のデータモデルは, DC が管理する永続データを外部仕様としてモデル化したものであり, その実現手段は規定しない. すなわち, DC のデータモデルは abstract interface である.

インタフェース

インタフェースは UML の *Interface* として表現され, DC, PC 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. パラメータの方向は `in`, `out` キーワードで指定される. 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. 従って操作パラメータの型は *DataType*, *PrimitiveType*, *Enumeration* の何れか, またはその *Collection* である[*9][*10]. 図 2.6 の例では, 操作 `IWSExecutionAgent::sysExecuteWS` の `in` パラメータ `service` と, 操作 `IAccessConfigMgt::getAccessibleWS` の `return` パラメータの *DataType* 型として, それぞれ `Service`, `WSInfo` が定義されている.

PC, DC は, おのおのが提供するインタフェース操作の機能を実現する. PC 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出来るが, 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出来ない. これによりコンポーネント間の結合が疎に保たれる.

[*9] 本論文では, クラス間の関連または操作パラメータの型に *Collection* を用いる場合, 順序無しの *Collection* に限定する.

[*10] このようなコンポーネント間のデータの値渡しのために定義されるデータ型を, Herzum et al. [3] では Business data type と呼んでいる.

整合性制約

各 DC のデータモデルには，問題領域に固有の整合性制約が定義される．データを図形として表現する UML コンポーネント図では，関連の多重度などの一部の基本的な制約のみ記述可能であり，一般的な整合性制約を記述する能力を備えていない．そこで，データモデルの制約を記述するために，OCL が広く用いられている．OCL は MOF ベースのモデルに対するナビゲーション言語であり，単一の DC 内の整合性制約は OCL 不変条件（invariant）として表現される．OCL を用いた形式的な記述によって，無効なオブジェクトモデルを排除して，データモデルの完成度を高めることができる[*11]．

Ackermann et al. [75]，Wahler [55]，Miliauskaitė [56]，Costal et al. [57] において，整合性制約の分類と，OCL を用いた記述パターンが提案されている．これらの文献における提案は，分類の観点や粒度が少しずつ異なる．一例として，Costal et al. [57] の分類を以下に示す．

- Uniqueness：データモデル中の *Class* のオブジェクトが属性値によって一意に識別されること（一意性）を表す制約．
- Recursive Association：*Class* 自身への再帰的な関連における，オブジェクトのリンク構造に関する制約．Irreflexive（非反射的），Symmetric（対称的），Antisymmetric（反対称的），Asymmetric（非対称的），Acyclic（非循環的）の構造が存在する．
- PathComparison：2 つの *Class* 間の異なるパス（ナビゲーションの列）から得られるオブジェクト集合の関係を表す制約．Inclusion（包含），Exclusion（排除），Equality（等価）の関係が存在する．
- ValueComparison：オブジェクトの属性が取り得る値を規定する制約．

図 2.7 に，一意性制約を表す不変条件の記述例を示す．`context` と `inv` は，それぞれ，UML モデル中の文脈と不変条件を宣言するキーワードである．図 2.7 の記述は，`AccessConfig` の `Project` クラスの全てのインスタンスが，属性 `id` の値によって一意に識別されることを表している．すなわち，`id` は `Project` のキー属性である．

DC 間の参照

単一の DC 内のデータの対応関係は，一般に *Class* 間の関連を用いて表現される．一方，異なる DC のデータモデルの間には直接の関連は存在しないため，DC 間に跨るデータの対応関係は，参照キーを介した間接的な参照として表現される．(Herzum et al. [3]，Cheesman et al. [47]，D'Souza et al. [50])．

```
1  context AccessConfig
2  inv: AccessConfig::Project.allInstances()->isUnique(id)
```

図 2.7. 一意性制約の記述

[*11] Briand et al. [54] は実証実験を通じて，OCL を用いた制約記述が，UML モデルの誤り検出・理解・メンテナンスの各作業の精度向上に寄与することを示した．

```
1  context AccessConfig::getAccessibleWS(projectID:String):Set(WSInfo)
2  post: let wsDefs:Set(WSDef)=
3    Project.allInstances()->select(i|i.id@pre=projectID).WSDef in
4    (wsDefs->isEmpty() implies result->isEmpty()) and
5    wsDefs->forAll(j|result->exists(k|
6      k.id=j.id@pre and k.version=j.WSImp.version@pre))
```

図 2.8. `AccessConfig::getAccessibleWS` の振舞いモデル

```
1  context WSStatusMgr::addAccessor
2    (wsDefID:String,wsImpVersion:String,projectID:String)
3  post: let wsInfos:Set(WSInfo)=WSInfo.allInstances()->select(m|
4      m.wsDefID=wsDefID and m.wsImpVersion=wsImpVersion) in
5    wsInfos->one(true) and
6    wsInfos->any(true).Accessor->exists(n|n.projectID=projectID)
```

図 2.9. `WSStatusMgr::addAccessor` の振舞いモデル

DC 間の参照の集合は，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，$cls_{src}.prop_{src}$ から $cls_{dst}.prop_{dst}$ への参照を表す．ここで，参照先の属性 $prop_{dst}$ は cls_{dst} のインスタンスを一意に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．参照元の属性 $prop_{src}$ は，cls_{dst} のインスタンスを指定する役割を持つ．

REF の要素は，$\langle\langle$`ref`$\rangle\rangle$ を付加した依存としてコンポーネント図に記載される．図 2.6 の例では，$\mathtt{r_1} = (\mathtt{Accessor}, \mathtt{projectID}, \mathtt{Project}, \mathtt{id})$，$\mathtt{r_2} = (\mathtt{WSInfo}, \mathtt{wsDefID}, \mathtt{WSDef}, \mathtt{id})$ の 2 つの参照がモデル化されている．これらの参照のように，$prop_{src}$ と $prop_{dst}$ は異なる名前を持つことが可能であるが，両者は同一の *PrimitiveType* 型であるものとする．

2.3.3 DC の振舞いモデル

DC の振舞いモデルは，OCL を用いて操作の振舞い仕様を宣言的に記述したモデルであり，操作を呼び出すための前提条件（事前条件）と，操作を実行することによる効果（事後条件）からなる．事後条件には，DC が管理するデータに関する CRUD 機能，すなわち Create（生成），Read（読み取り），Update（更新），Delete（削除）の内容が，オブジェクト，属性，リンク，操作パラメータの間に成り立つ論理式で記述される．

OCL の言語仕様では，表 2.3 中の各型ごとに利用可能な演算を定めている．例として，等価演算（=），数値演算（+，−，∗，/），論理演算（`and`，`or`，`not`，`implies`），*Collection* に対する演算（`size`，`includes`，`isEmpty`），量化演算（`forAll`，`exists`）が挙げられる．これらの演算の他に，部分式を表す変数を定義する `let` 表現，`if-then-else-endif` 表現などの構文要素を組み合わせて，事前・事後条件が構成される．

図 2.8 と図 2.9 に，例題の `AccessConfig` の操作 `getAccessibleWS` の振舞いモデルと，

WSStatusMgr の操作 addAccessor の振舞いモデルをそれぞれ示す．post は，事後条件を宣言するキーワードである．“.”（ドット）と “->” は，それぞれ，オブジェクト間のリンクのナビゲートと，*Collection* のプロパティに対するアクセスを表現する．キーワード result は，操作の戻り値を表す．

getAccessibleWS の事後条件は，操作パラメータ projectID の値を属性 id に持つような Project オブジェクトにリンクされる WSDef のオブジェクト集合を wsDefs として（2–3 行目），wsDefs が空ならば操作の戻り値が空であり（4 行目），wsDefs の全ての要素 j について，j の属性 id の値を属性 id に持ち，j にリンクされる WSImp オブジェクトの属性 version の値を属性 version に持つような，result の要素 k が存在する（5–6 行目）ことを表現している[*12]．

addAccessor の事後条件は，操作パラメータ wsDefID，wsImpVersion の値を属性 wsDefID，wsImpVersion にそれぞれ持つような WSInfo オブジェクトの集合を wsInfos として（3–4 行目），wsDefID の要素がただ 1 つ存在し（5 行目），その要素にリンクされる Accessor オブジェクトの集合の中に，パラメータ projectID の値を属性 projectID に持つ要素 n が存在することを表現している．

2.3.4　PC の振舞いモデル

PC の振舞いモデルは，PC 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り，ビジネスプロセスや，プログラムの制御とデータの流れを表すために，広く用いられている．本手法が扱う *Activity* の要素を図 2.10 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，true（*Boolean*），512（*Integer*），‘PRJ0023’（*String*）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，PC 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．

UML 仕様（OMG [22]）には，図 2.10 の要素の他に，イベントの送受信を表す *SendSignalAction* と *AcceptEventAction*，処理フローを分割して記述するための *ActivityPartition* 等が定義されている．我々の仕様モデルでは，コンポーネント間の相互作用の手段を操作呼び出しに限定している．また，PC の操作ごとに 1 つの処理フローを記述することを想定している．従って，本手法ではこれらの要素を扱わない．

図 2.11 と図 2.13 に，例題の 2 つの PC（WSExecutionAgent，WSProvider）の操作

[*12] @pre は事後条件中で操作実行前の値を参照する OCL のキーワードである．

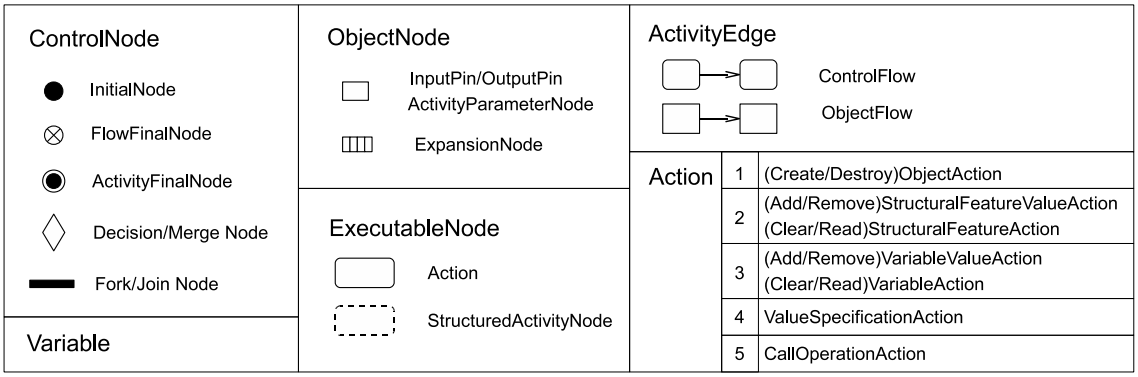


図 2.10. *Activity* の要素

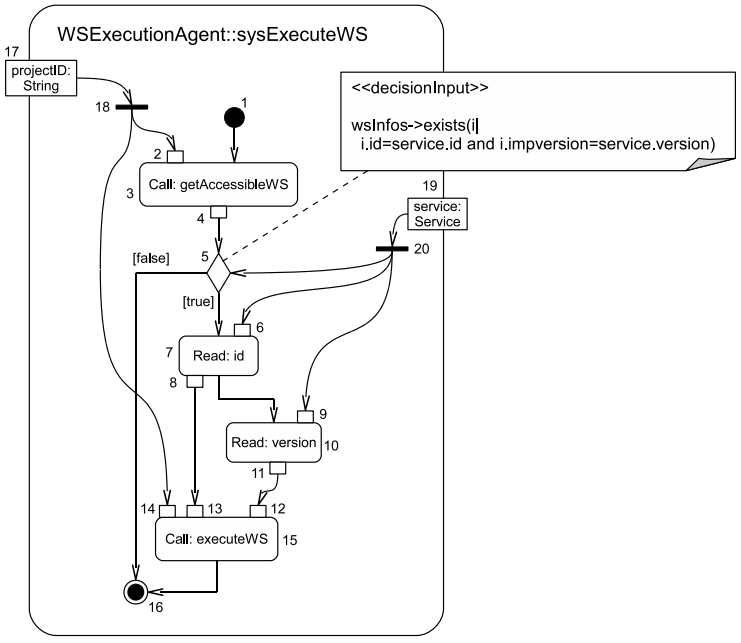


図 2.11. `WSExecutionAgent::sysExecuteWS` の振舞いモデル

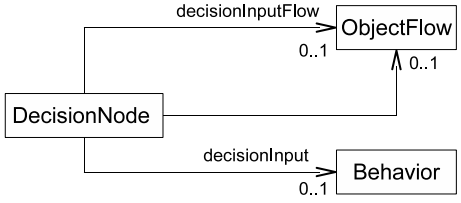


図 2.12. *DecisionNode* のメタモデル

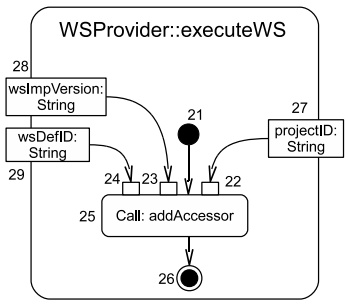


図 2.13. `WSProvider::executeWS` の振舞いモデル

`sysExecuteWS`，`executeWS` の振舞いモデルを示す．説明の都合上，各ノードには識別番号（1–29）を付加している．

`sysExecuteWS` の *Activity* は，番号 3 の *CallOperationAction* において，`AccessConfig` のデータアクセス操作 `getAccessibleWS` を呼び出して，番号 2 の *InputPin* で表されるプロジェクトが利用可能な WebService の情報を，番号 4 の *OutputPin* として取得する．その際に，番号 2 の *InputPin* には，番号 18 の *ForkNode* を介して番号 17 の *ActivityParameterNode* の値が渡される．そして，番号 5 の *DecisionNode* において，番号 20 の *ForkNode* を介して伝達される番号 19 の *ActivityParameterNode* の内容が，番号 4 の *OutputPin* の情報に包含されることの判定を行う．*DecisionNode* には *Behavior* が設定されており（図 2.12），*ObjectFlow* を通じて入力されるデータを引数として，*Behavior* を呼び出すことによって，判定が行われる．判定結果が真の場合には，番号 7，10 の *ReadStructuralFeatureAction* で，番号 19 の *ActivityParameterNode* から渡される `Service` オブジェクトの属性 `id`，`version` をそれぞれ読み出して，番号 15 の *CallOperationAction* において，`WSProvider` の操作 `executeWS` を呼び出す．一方，番号 5 の判定結果が偽の場合には，番号 16 の *ActivityFinalNode* に制御が移り，処理が終了する．

`executeWS` の *Activity* は，番号 25 の *CallOperationAction* において，`WSProvider` のデータアクセス操作 `addAccessor` を呼び出す．番号 22，23，24 の *InputPin* には，番号 27，28，29 の *ActivityParameterNode* の値が渡される．

2.4　振舞いの整合性判定

ソフトウェアの不具合の発見・修正コストは，開発の後工程になるにつれて急激に増大することが知られている（Boehm et al. [58, 59]）．従って，開発の早期にソフトウェアの品質を確保することが，システム開発を成功させるための鍵である．UML/OCL ベースの情報システムの CBD においては，仕様モデルに基づいてコンポーネントの実装・利用を行うため，仕様定義段階で UML/OCL モデルの不整合を検出・除去することが極めて重要である．

本節では，2.3.2 項で定義した DC 間に跨る参照の整合性を確保するための，DC と PC の振舞いの整合性について論じる．2.4.1 項で，モデルの整合性に関する一般的な概念を述べ，

整合性の種類と判定条件，不整合への対処プロセス，整合性の判定方法について説明する．2.4.2 項において，参照整合性と DC，PC の振舞いの整合性の関係を述べる．2.4.3 項で，例題における振舞いの整合性の具体例を示し，2.4.4 項において，振舞いの整合性判定の難しさを説明する．2.4.5 項において，仕様モデルの不整合が引き起こす開発の手戻りについて述べる．

2.4.1 モデルの整合性

ソフトウェアシステムを構成するモデル群は，様々なステークホルダによって定義・利用され，システムの異なる側面について，個別の記法を用いてモデルの記述が行われる．従って，複数のモデルの間に不整合が生じる可能性がある．Spanoudakis et al. [60] は，モデルの不整合（Inconsistency）を次のように定義した．

> “A state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are not jointly satisfiable.”（異なるソフトウェアのモデルにおいて重複する複数の要素が，システムの側面について充足不能な表明を行っている状態.）

不整合が存在しない無矛盾な性質を，整合性（Consistency）と呼ぶ．Lucas et al. [61], Usman et al. [62], Engels et al. [63] 等の UML モデルの整合性に関する多くの文献では，上記の Spanoudakis らの不整合の定義が用いられている．本論文でも，この定義に従って仕様モデルの整合性を扱う．

整合性の種類

モデルの整合性は，対象とするモデル群の関係によって分類することができる．従来の研究で提案されている，整合性の一般的な種類を以下に挙げる[*13]．

- Horizontal (intra-model) consistency: 対象システムの異なる側面を記述するモデル間の整合性（Huzar et al. [65]）．
- Vertical (inter-model) consistency: 抽象度の異なるモデルの間の整合性（Huzar et al. [65]）．
- Evolution consistency: 同一モデルの異なるバージョン間の整合性（Yang [66]）．
- Syntactical consistency: メタモデルで定義された抽象構文に対する，モデルの統語的な整合性（Engels et al. [63]）．
- Semantic consistency: 対象システムの特定の側面に関する，モデル間の意味的な無矛盾性（Engels et al. [63]）．

[*13] 特定のモデリングの枠組みに固有の整合性も存在する．例えば Engels et al. [64] では，オブジェクト指向モデルの継承関係に固有の Observation consistency と Invocation consistency を取り上げている．前者はスーパークラスとサブクラスの振る舞いモデル間の整合性を，後者はオブジェクトの置換可能性を表す．

Horizontal consistency は，異なる側面を記述する複数のモデルの組合せが，対象システムを矛盾なく表現することを要求する．UML モデルにおける Horizontal consistency の例として，静的な側面を記述するクラス図と，動的な側面を記述するシーケンス図の間の整合性が挙げられる．

Vertical consistency は，モデルベースの開発においてモデルの抽象度を段階的に詳細化する際に，抽象モデルに対して，そのモデルを詳細化した具象モデルが無矛盾であることを要求する．Vertical consistency の例として，設計レベルのモデルの，分析レベルのモデルに対する整合性が挙げられる．

Evolution consistency は，異なる側面を記述するモデル間の Horizontal consistency を保ちながら，モデルのバージョンアップを行う場合に必要となる．例えば，UML クラス図とシーケンス図の間の Horizontal consistency を保ちながら，クラス図のバージョンを v1 から v2 に更新する場合に，v1 と v2 の間の Evolution consistency が必要である．

Syntactical consistency は，メタモデル階層において，或る階層のモデルが，隣接する上位階層のメタモデルに対して整形式であることを要求する．例えば，UML オブジェクト図中のオブジェクト間のリンクが，UML クラス図中の多重度の定義に従う，などの例が挙げられる．

Semantic consistency は，Syntactical consistency を前提として，モデル間により厳密な制約を課す．例えば，UML のシーケンス図とコミュニケーション図に記述されているオブジェクト間のイベント交換が同一の生起順序に従う，等の例が挙げられる．

1 つの整合性が，複数の種類の特徴を兼ね備えるケースも存在する．前述のシーケンス図とコミュニケーション図の間の，イベント生起順序に関する Semantic consistency の例を用いて説明する．シーケンス図は時系列の観点から，コミュニケーション図は構造の観点から，それぞれオブジェクト間の相互作用を記述する．従って，この例における整合性は Horizontal consistency の特徴も同時に備えている．

情報システムに対する要求を概念モデルとして正確に表現することが，システムの品質の確保に大きく寄与する（Moody [67]）．ソフトウェアのモデルは，対象ドメインに対する人の心象を抽象化した表現である．従って，心象を正しく反映したシステムを構築するためには，モデルに対する記述者の意図（Intent）や利用者の解釈と，モデルの言語上の表現が無矛盾であることが重要である．本論文では，前述の 5 種類のソフトウェアモデルの間の整合性の他に，モデル記述者の意図やモデル利用者の解釈と，ソフトウェアモデルの間の整合性を扱う．

- Intent consistency: モデル記述者の意図，モデル利用者の解釈，モデルの言語上の表現の間の整合性．

整合性の判定条件

不整合を検出するために，整合性の有無を判定する条件（rule）が必要である．Spanoudakis et al. [60] は，判定条件の分類を次のように定義した．

- Well-formedness rule: Syntactical consistency に関して，モデルが整形式であること

を判定する条件.

- Description identity rule: モデル間の重なり合う要素を特定する識別情報が存在することを判定する条件.
- Application domain rule: ビジネスルール等の，問題領域に固有の整合性の判定条件.
- Development compatibility rule: 異なる複数の抽象モデルを詳細化して，1 つの具象モデルを構築することの可能性を判定する条件.
- Development process compliance rule: ソフトウェア工学の慣習や標準に対する，モデルの適合を判定する条件.

整合性の判定を機械的に実行するためには，判定条件の形式的な表現が必要である．UML モデルの制約記述には OCL が広く用いられているが，OCL で表現することができない判定条件を持つ整合性も存在する．例えば，Engels et al. [63] で扱われている，複数の UML-RT 状態遷移モデルの間のプロトコル整合性は，判定条件を OCL で記述することができない．Engels et al. [63] は，この問題に対して，プロセス代数の CSP を用いて判定条件を記述する方法を提案している．

一般にモデル記述者の意図そのものは形式的に表現されないため，Intent consistency の判定条件を明確に定義することは難しい．この問題に対して，Salay [68] は UML モデルの記述者の意図を表現するためのモデリング言語（Macromodel language）を提案している．

不整合への対処プロセス

仕様モデル間の不整合は，仕様モデルの欠陥に起因する矛盾であり，システムの安全性や信頼性などの品質を低下させる．特にモデルベースの CBD では，仕様モデルに基づいてコンポーネントの実装が行われるため，仕様モデルの欠陥が実装に伝播する前に，仕様定義の段階で不整合を検出・除去する必要がある．

Finkelstein et al. [69]，Nuseibeh et al. [70] は，不整合への対処（Inconsistency Management）を行うためのフレームワークを，それぞれ独立に提案した．Spanoudakis et al. [60] は，それら 2 つのフレームワークを統合して，不整合への対処プロセスを定義した．Spanoudakis et al. [60] のプロセスは，以下の活動から構成される．

- Detection of overlaps: ソフトウェアモデル間の重複要素を特定する.
- Detection of inconsistencies: 整合性の判定条件を用いて，不整合を検出する.
- Diagnosis of inconsistencies: 検出した不整合について，不整合の根拠を示すモデル要素，不整合の原因，システムに及ぼす影響を特定する.
- Handling of inconsistencies: 不整合に対するアクションの特定，アクションのコストと効果の評価，不整合を解消しない場合のリスクの評価，実行するアクションの選択，の各々を実施する.
- Tracking: 上記 4 つ（Detection of overlaps/Detection of inconsistencies/Diagnosis of inconsistencies/Handling of inconsistencies）の活動内容を後学のために記録する.
- Specification and application of an inconsistency management policy: 他の活動を実

施する上での規則を定義する．

本研究の主題は，Detection of inconsistencies の活動を実施するための，整合性の判定技術に関わる．

整合性の判定方法

Detection of inconsistencies における整合性の判定に関して，次の 4 通りのアプローチが存在する（Spanoudakis et al. [60]）．

- 論理学に基づく方法: 定理証明技術を用いて推論を行い，論理式で記述された条件式の真偽を判定する．
- モデル検査に基づく方法: 有限オートマトンで表現されたモデルに対して，モデル検査アルゴリズムを用いて，時相論理式で記述された条件の成立を判定する．
- 専用の解析を行う方法: 問題に特化した解析アルゴリズムを用いて，条件の成立を判定する．多くの場合，解析は自動化される．
- 人手による検査: 利害関係者が人手でモデルを検査して，整合性を判定する．

各アプローチには，それぞれ利点と制限が存在し，モデルや整合性の判定条件の性質に合わせて適切なアプローチを選択する必要がある．モデルや判定条件の記述が非形式的な場合には，人手による検査が唯一の選択肢である．一方，形式的なモデルと判定条件に対しては，その他のアプローチ（論理学に基づく方法，モデル検査に基づく方法，専用の解析を行う方法）によって，より厳密かつ効率的に整合性を判定できる可能性がある．

2.4.2　参照整合性と振舞いの整合性

2.3.2 項で述べた DC 間に跨る参照の整合性を保証するために，DC，PC の振る舞いの整合性が必要であることを説明する．

図 2.14 は, DC と PC の構造モデル, 振舞いモデル, オブジェクトモデルの関係を示す例である．$pc \in$ PC の振舞いモデルは，$dc_{src}, dc_{dst} \in$ DC の操作を利用する．dc_{src} と dc_{dst} の振舞いモデルは, 各々のデータアクセス操作の仕様を定義する．$objm_{src}$, $objm_{dst}$ は, dc_{src}, dc_{dst} の構造モデルに従うオブジェクトモデルであり，図中のステレオタイプ ⟨⟨`instance-of`⟩⟩ は，構造モデルとの間の統語的な整合性（Syntactical consistency）を示している．

dc_{src} と dc_{dst} の間に，DC 間に跨る参照 $r = (cls_{src}, prop_{src}, cls_{dst}, prop_{dst}) \in REF$ が存在する場合を考える．ただし $cls_{src}, cls_{dst} \in Class$ は，それぞれ dc_{src}，dc_{dst} の構造モデルで定義されたクラスである．参照 r は，構造モデルと $objm_{src}$，$objm_{dst}$ の組合せについて，意味的な整合性（Semantic consistency）を要求する．参照整合性の判定条件は次の論理式で表現される．

$$\forall o_{src} \in cls_{src}(\exists o_{dst} \in cls_{dst}(o_{src}.prop_{src} = o_{dst}.prop_{dst}))$$

$objm_{src}$ と $objm_{dst}$ の組合せは，dc_{src}，dc_{dst} の操作の振舞いモデルと，それらの操作を利

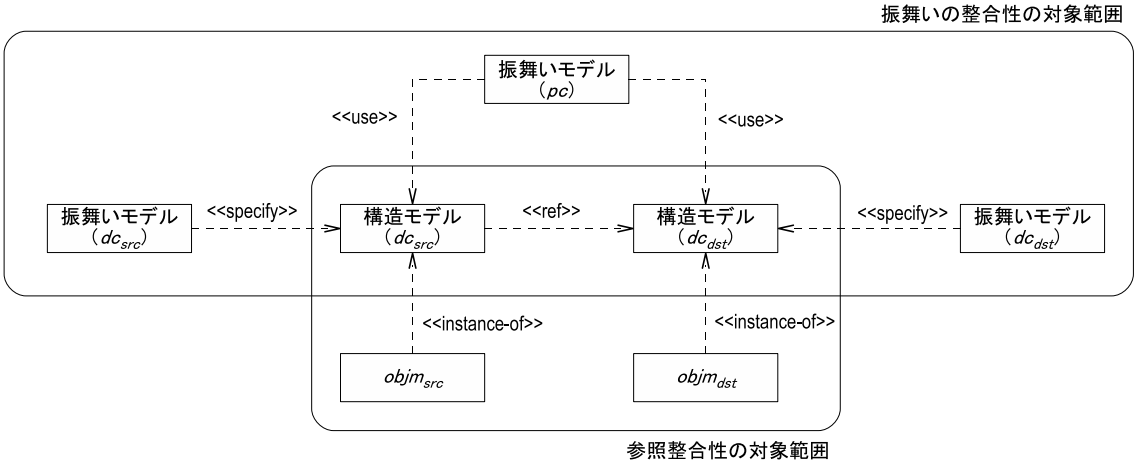


図 2.14. 構造モデル，振舞いモデル，オブジェクトモデルの関係を示す例

用する PC の振舞いモデルの組合せによって生成される．従って，生成され得る全てのオブジェクトモデルの組合せについて，r の参照整合性を保証するためには，DC の操作の振舞いモデルと，それらの操作を利用する PC の振舞いモデルについて，振舞いの整合性を判定する必要がある．図 2.14 の例における，参照整合性と振舞いの整合性のそれぞれの対象モデル群を，図中に示す．

振舞いの整合性は，次の 2 種類の整合性に分けて考える必要がある．

- 個々の振舞いモデルの言語上の表現と，モデル記述者の意図，モデル利用者の解釈の間の整合性（Intent consistency）．
- 異なる振舞いモデルの間の整合性（Horizontal consistency）．

異なる振舞いモデルの整合性を議論するためには，個々の振る舞いモデルの表現が，モデル記述者の意図を正確に反映したものでなければならない．従って，Intent consistency は Horizontal consistency の前提となる．

本論文では，DC の振舞いモデルにおけるデータアクセスに関する整合性（Intent consistency）と，PC の振舞いにおける DC のデータアクセス操作の利用手順の整合性（Horizontal consistency）を扱う．これら 2 つの整合性について説明する．

整合性 1 : DC のデータアクセスの整合性

DC の振舞いモデルでは，データアクセス操作の振舞いが，OCL を用いて宣言的に記述される．モデル記述者は，操作の実行を通じたデータへのアクセスを厳密に記述し，モデル利用者，すなわち PC の振舞いモデルの記述者は，それらの記述内容を正確に理解する必要がある．

宣言的な仕様記述言語には，操作の振舞いを実現方法に依らずに記述できる利点があるが，振舞いを一意に特定できない “underspecified”な記述を受容する．このことから，事後条件に明示的に記述されないモデル要素の状態が確定しない，というフレーム問題が生じ

る（Wieringa [5]）．従って，DC の振舞いモデルの表現と，記述者の意図，利用者の解釈の間に不整合が生じる恐れがある．そこで本論文では，DC のデータアクセスに関する Intent consistency の判定方法の研究に取り組む．

なお，PC の振舞いモデルでは，記述者が意図する処理手順が，UML Activity の命令的なフロー記述としてそのまま表現されるため，フレーム問題は発生せず，Intent inconsistency の恐れは少ない．

整合性 2 : PC の振舞いにおける DC の操作の利用手順の整合性

DC 間に跨る参照の整合性を保証するためには，参照先と参照元の DC のオブジェクト状態を同期させる必要がある．そのような同期を行うためのアーキテクチャ上のアプローチとして，以下の選択肢が存在する．

1) 参照先または参照元の DC が同期を行う．
2) DC のデータアクセス操作を利用する PC が同期を行う．
3) 複数の DC の同期管理を行う専用のコンポーネントを設ける．

アプローチ 1) では，DC 間に直接の依存が必要となり，場合によっては依存関係の循環が発生する可能性がある．通常，情報システムの CBD では DC の再利用性を保つために，個々の DC は他への依存を持たない独立したコンポーネントとして定義され，自身が管理するデータの整合性のみ保証する．従って，このアプローチが採られることは少ない．

アプローチ 2) では，PC の振舞いの中で，参照先と参照元の DC のデータアクセス操作を適切な手順で利用することによって，オブジェクト状態の同期を行う．このアプローチは，オブジェクト状態の同期を行う責務が局所化されず，DC を利用する各 PC の振舞いに責務が分散する欠点を持つ．しかし，参照整合性を保証する現実的な解法として，多くの情報システムの開発で採用されている．

アプローチ 3) は，複数の異種データベースの間の整合性の問題（Bright et al. [71]）に対する，統合管理方法（Ahmed et al. [72]，Litwin et al. [73]）の類比である．オブジェクト状態の同期を行う責務が局所化されるが，システム内の全ての DC の仕様モデルの進化に追従して，継続的に整合性を矛盾なく統合管理するコストが必要となる．

本論文では，アプローチ 2) によって DC のオブジェクト状態の同期が行われる場合を想定して，PC の振舞いにおける，DC のデータアクセス操作の利用手順の Horizontal consistency の判定を行う方法の研究に取り組む．

2.4.3　例題における振舞いの整合性

2.3 節の例題を用いて，2.4.2 項で導入した整合性 1（DC のデータアクセスの整合性）と整合性 2（PC の振舞いにおける DC の操作の利用手順の整合性）の具体例を示す．

まず，整合性 1 について述べる．図 2.8 の `AccessConfig` の操作 `getAccessibleWS` の事後条件は，次の内容を表現している．

- 入力パラメータ projectID の値をキー属性 id に持つ Project オブジェクトが存在することと，集合 result が非空であることが，等価である．すなわち，projectID から特定される Project オブジェクトの存在の照合を行い，その結果を result の値として与える．
- 入力パラメータ projectID から特定される Project オブジェクトからのリンクを辿って到達する WSDef，WSImp オブジェクトの属性 id，version の値が，戻り値 result の要素の属性として得られる．すなわち，ある条件に合致する WSDef，WSImp オブジェクトの属性 id，version の値の取得を行う．

同様に，図 2.9 の操作 addAccessor の事後条件は，次の内容を表現している．

- 操作実行後に，入力パラメータ wsDefID と wsImpVersion の値を属性に持つ WSInfo オブジェクトがただ 1 つ存在する．すなわち，wsDefID，wsImpVersion から WSInfo オブジェクトの対応する属性への値の代入を行う可能性がある[*14]．
- 操作実行後に，入力パラメータ projectID の値を属性 projectID に持つ Accessor オブジェクトが存在する．すなわち，projectID から Accessor オブジェクトの属性 projectID への値の代入を行う可能性がある

getAccessibleWS，addAccessor の振舞いモデルの記述者は，自身が意図するデータアクセスを振舞いモデルに正確に表現する必要がある．また，これらの振舞いモデルの利用者は，振舞いモデルの意味を厳密に解釈して，上述した照合，取得，代入のデータアクセスを把握する必要がある．

続いて，整合性 2 について述べる．例題の合成モデルは，2.4.2 項で述べたアーキテクチャ上のアプローチ 2) に従って，PC の振舞いの中で，DC のデータアクセス操作を適切な手順で利用することによって，DC 間に跨る参照の整合性を保つ．

図 2.15 を用いて，例題の合成モデルの処理手順が，参照 $\mathtt{r}_1$ の整合性を保つことを説明する[*15]．sysExecuteWS は，番号 17 の操作パラメータ projectID の値を番号 2 の *InputPin* に伝搬し（手順 A），その値を引数 projectID として AccessConfig の操作 getAccessibleWS を呼び出し（手順 B），projectID の値を属性 id に持つ Project オブジェクトの存在を照合する．そして番号 17 の操作パラメータ projectID の値を番号 14 の *InputPin* に伝搬し（手順 C），その値を引数 projectID として WSProvider の操作 executeWS を呼び出す．executeWS は，パラメータ projectID の値を番号 22 の *InputPin* に伝搬し（手順 D），その値を引数 projectID として WSStatusMgr の操作 addAccessor を呼び出して，Accessor オブジェクトの属性 projectID に代入する（手順 E）．合成モデルは，手順 E の代入に先立って手順 B で照合を行うことによって，代入が参照 $\mathtt{r}_1$ の整合性を破壊しない安全な文脈で実行

[*14] 操作実行前に，入力パラメータ wsDefID の値を属性に持つ WSInfo オブジェクトが存在しない場合に，事後条件を充足するために，ある WSInfo オブジェクトの属性にパラメータ wsDefID の値が代入されるはずである．

[*15] 説明の都合上，図 2.15，図 2.16 に実行順を示すアルファベット記号（A–E，F–J）と補助線を付加してある．

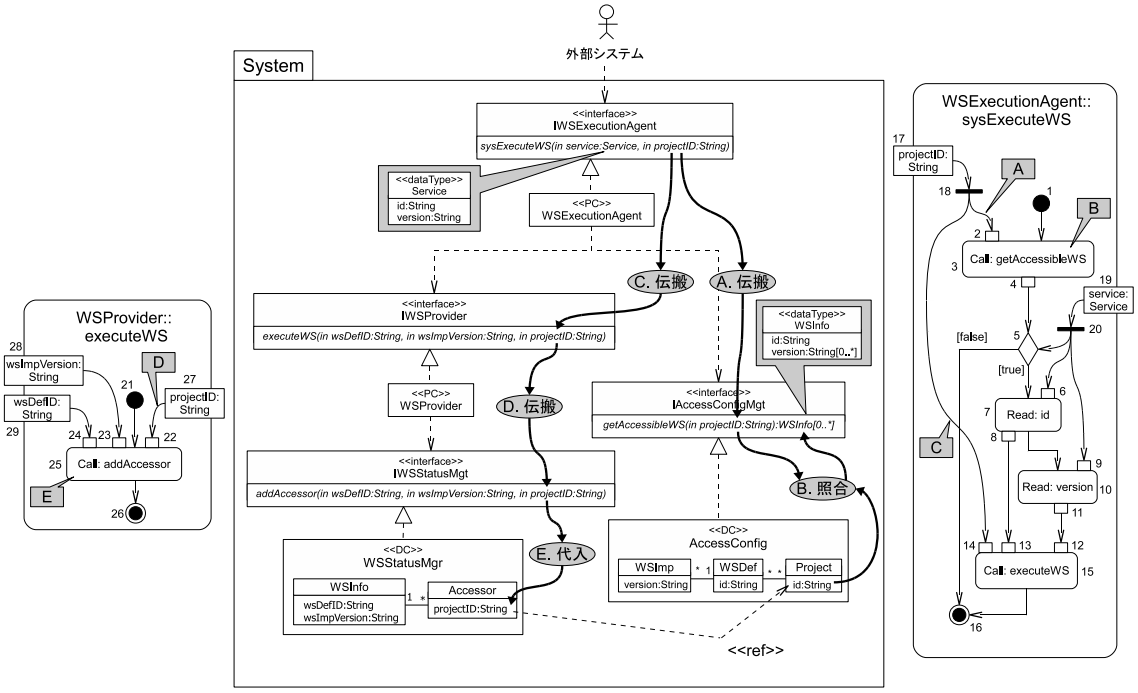


図 2.15. 振舞いの合成と DC 間の参照 $\mathtt{r}_1$

されることを保証している．

同様に，図 2.16 を用いて，合成モデルの処理手順が参照 $\mathtt{r}_2$ の整合性を保つことを説明する．`sysExecuteWS` は，`AccessConfig` の操作 `getAccessibleWS` を呼び出し，`WSDef` オブジェクトの情報を取得して（手順 F），番号 19 の `Service` オブジェクトの属性 `id` の値が，取得した情報に包含されることの判定を行う（手順 G）．そして，`Service` オブジェクトの属性 `id` の値を番号 13 の *InputPin* に伝搬し（手順 H），その値を引数 `wsDefID` として `WSProvider` の操作 `executeWS` を呼び出す．`executeWS` は，パラメータ `wsDefID` の値を番号 24 の *InputPin* に伝搬し（手順 I），その値を引数 `wsDefID` として `WSStatusMgr` の操作 `addAccessor` を呼び出して，`WSInfo` オブジェクトの属性 `wsDefID` に代入する（手順 J）．合成モデルは，手順 J の代入に先立って，手順 F で参照先の属性値を取得して手順 G で代入値の包含判定を行うことによって，代入が参照 $\mathtt{r}_2$ の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．

`sysExecuteWS`，`executeWS` の振舞いモデルの記述者は，参照 $\mathtt{r}_1$，$\mathtt{r}_2$ の整合性を保つために，上述した A–E，F–J のような適切な手順で DC のデータアクセス操作を利用する，整合性のとれた振舞いモデルを記述する必要がある．また，モデルの合成を行うインテグレータは，振舞いモデルの整合性を確認する必要がある．

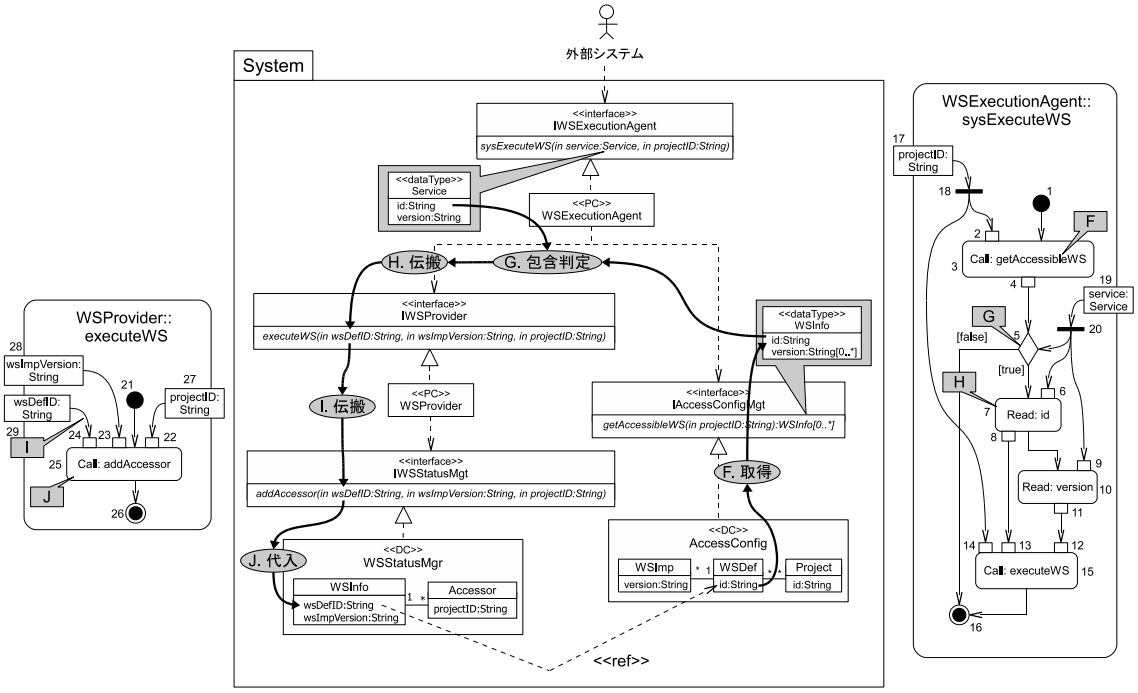


図 2.16. 振舞いの合成と DC 間の参照 $\mathbf{r}_2$

2.4.4 振舞いの整合性判定の難しさ

2.4.2 項で述べたように，DC 間に跨る参照の整合性を保証するために，次の 2 つの振舞いの整合性を判定する方法が必要である．

整合性 1： DC のデータアクセスの整合性
整合性 2： PC の振舞いにおける DC の操作の利用手順の整合性

これらの振舞いの整合性は，いずれも判定を正確に行うことは容易ではない．本節では，2 つの振舞いの整合性を判定する難しさを説明する．

まず，整合性 1 の判定の難しさについて述べる．判定の難しさの要因として，OCL の言語特性と，人手による判定の誤りが挙げられる．

OCL の言語特性

OCL は，集合論と述語論理に基づく数学的な意味論を有すると同時に，利用者の利便性を重視して，これらの数学的な概念を簡便に表すための ASCII 記法を備えている（Warmer et al. [74]）．それでもなお，複雑な制約を OCL で表現する場合に，記述者の意図と実際の記述内容の間に相違が生じたり，記述内容が誤って解釈される恐れがある（Ackermann et al. [75], Correa et al. [76]）．

特に，Cabot [77]，Inoue et al. [78] で言及されているように，宣言的な仕様記述言語であ

る OCL には，データに対する代入，照合，取得のアクセスを表現する専用の演算子が存在しないため，アクセスの記述が暗黙的であり，それらを正確に識別することは容易ではない，という問題がある．たとえば，2.3.3 項の DC の振舞いモデルの説明において，例題の操作 `getAccessibleWS` と `addAccessor` の中で照合，取得，代入のアクセスが行われることを説明したが，これらのアクセスは何れも等価演算子 “`=`” を用いて記述されている．3 者を正確に識別するためには，振舞いモデルの意味を正しく解釈する必要がある．また OCL では，オブジェクトの消滅を明示的に表現する演算子が存在しない．

さらに，OCL の論理式は，文脈依存性・非決定性・不完全性の性質を有する．

文脈依存性： 論理式を構成する個々の OCL 表現の内容の解釈は，その表現の文脈に依存する．例として，表現 `X` が或る性質の存在を表す場合に，“`not X`”は性質の不在を表す．

非決定性・不完全性： OCL は三値論理に基づいており，一般に論理式は非決定性と不完全性を有する．例として，表現 “`X or Y`”の値が `true` の場合に，`X` と `Y` のいずれが `true` の値をとるか決定することは出来ない（非決定性）．また `X` の値が `true` である場合に `Y` の値を特定することが出来ない（不完全性）．

例題の `getAccessibleWS` の振舞いモデルには，表現 “`wsDefs->isEmpty() implies result->isEmpty()`” が含まれる（図 2.8 の 4 行目）．この表現は “`not wsDefs->isEmpty() or result->isEmpty()`” と等価であるが，上記の不完全性に関する理解が不足している場合には，“`result->isEmpty()` が真のときに `wsDefs->isEmpty()` は真となる” と誤って解釈する，後件肯定の誤謬がおきる恐れがある．

そして 2.4.2 項で述べたように，宣言的な仕様記述言語に共通の問題として，OCL の記述内容の解釈において，振舞いを一意に特定できないフレーム問題が存在する．

人手による判定の誤り

2.4.1 項で述べたように，一般にモデル記述者の意図そのものは形式的に表現されないため，Intent consistency の判定条件を明確に定義することが出来ない．従って，OCL で記述された DC の振舞いモデルについて，データアクセスに関する整合性 1 の判定を機械的に行うことは困難であり，何らかの方法によって人手で判定を行う必要がある．しかし，人手で判定を行う場合に，OCL の言語特性が判定の誤りを招く恐れがある．判定を正確に行うための手法やツールが必要である．

続いて，整合性 2 の判定の難しさについて述べる．判定の難しさの要因として，判定条件の複雑性と，合成モデルの複雑性が挙げられる．

判定条件の複雑性

PC の振舞いモデルにおける DC の操作の利用手順の整合性を判定するために，振舞いの合成モデルにおけるデータアクセスの手順と，データ伝搬の両方を考慮する必要があり，判定条件が複雑である．

データアクセスの手順： 参照先と参照元の DC のオブジェクト状態の参照整合性を保つためには，振舞いの合成モデル中で行われるデータアクセスの手順が重要である．そして，参照整合性を実現するアクセス手順は一様ではなく，参照に応じて異なる手順が採られる．例えば，例題の合成モデルは，参照 $\mathtt{r}_1$ の整合性を満たすために，参照先のデータの照合を行う（図 2.15 の手順 B）．一方，参照 $\mathtt{r}_2$ の整合性を満たすためには，参照先のデータの取得を行う（図 2.16 の手順 F）．

データ伝搬： PC の *Activity* は，DC のデータアクセス操作の値渡しのパラメータを介して，DC 内のデータにアクセスする．従って，異なる DC のデータ間の対応関係を確認するためには，図 2.16 の手順 H のようなオブジェクトの属性値の伝搬を辿って，データ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．

合成モデルの複雑性

大規模なシステムでは，参照先と参照元の DC の操作呼び出しが，異なる PC の振舞いモデルの中で行われるケースがある．そこで，図 2.15 や 図 2.16 のような，複数の *Activity* からなる合成モデルについて，整合性の判定を行う必要がある．人手によって，多くの *Activity* の内容を詳細に把握して判定を行うことは難事である．また，*Activity* の組み合わせはシステムの機能に対応して数多く存在する．そのような多数の組み合わせを，人手で網羅的かつ正確に判定することは困難である．

数多くの *Activity* の組み合わせについて，整合性 2 の判定を行うためには，複雑な判定条件を形式的に定義し，厳密かつ効率的に判定を実行するための手法やツールが必要である．

2.4.5 仕様モデルの不整合と開発の手戻り

人手による検査に頼って仕様モデルの整合性を確保する開発プロセスでは，2.4.4 項で述べた，振舞いの整合性判定の難しさによって，検査の誤りによる開発工程の手戻りが起こりがちである．図 2.17 に示す典型的な開発プロセスを用いて説明する．このプロセスでは，次の各工程を順に実施して仕様モデルの定義が行われる．

DC 仕様定義：

DC の構造モデルと振舞いモデルが定義される．仕様レビュー等の人手による検査で，整合性 1 の判定が行われた後に，DC の仕様モデルが決定される．

PC 仕様定義：

DC の仕様モデルに基づいて，PC の構造モデルと振舞いモデルが定義される．仕様レビュー等の人手による検査で，単体の *Activity* について整合性 2 の判定が行われた後に，PC の仕様モデルが決定される．

モデルの合成と検証（人手）：

インテグレータによって，DC と PC の仕様モデルの合成が行われる．合成後のシステム全体のモデルについて，仕様レビュー等の人手による検査で整合性 2 の判定

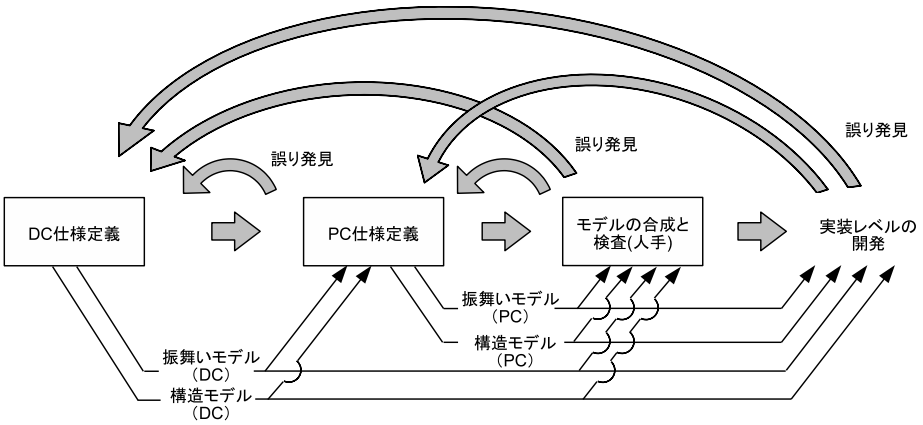


図 2.17. 整合性判定の誤りと開発の手戻り

が行われる．

そして引き続き，各コンポーネントの仕様の実現を目的として，コンポーネントの実装，合成，テスト，配備を行う実装レベルの開発フェーズへと進む．

しかし実際には，各工程の人手による整合性判定の作業において漏れや誤りが起きて，欠陥を含んだ仕様モデルが次工程に引き渡されてしまう可能性がある．そのようなケースでは，図 2.17 に示すように，仕様モデルの誤りが後工程で発見された時点で，前工程に戻って改めて仕様モデルの定義を行う手戻りが発生する．そして誤りの発見が後工程になる程，手戻りのコストは増大する．

このような仕様モデルそのものの欠陥がもたらす手戻りを防ぐためには，仕様定義の各工程において，仕様モデルの整合性を確保することが重要である．そのためには，2.4.4 項で述べた振舞いの整合性判定のむずかしさに対処して，整合性 1 と整合性 2 を正確かつ効率的に判定する方法が求められる．

仕様モデルそのものの欠陥の他に，仕様定義の入力となる要件の抽出漏れや齟齬が発端となり，仕様モデルの変更を余儀なくされる場合もある．一度の要件定義工程で全ての要件を完全に確定することは困難であり，要件の抽出漏れや齟齬の発生は避けられない．そのため近年では，要件の抽出漏れ・齟齬を早期に解消することを目指して，手戻りの影響が大きい伝統的なウォーターフォール型の開発プロセスの代わりに，RUP: Rational Unified Process（クルーシュテン [79]）等の反復型の開発プロセスや，プロジェクトの関係者が意思疎通を図りながら適応的に開発を行う，XP: eXtreme Programming（Beck [80]）等のアジャイル型の開発プロセスを採用する事例が増えつつある．

反復型・アジャイル型の開発プロセスでは，あるイテレーションの設計以降の工程で仕様モデルの欠陥が発見された場合に，後続のイテレーションにおいて，仕様モデルの修正を行い，

それに伴う設計・実装の変更とテストを実施する[*16]．もしイテレーションの度に仕様モデルの誤りが発覚すれば，それらに対応するための修正作業に追われて，イテレーションの繰り返しが収束しなくなる恐れがある．従って，これらの反復型の開発プロセスを採る場合においても，仕様モデルの整合性を確保するための方法が必要である．

2.5 関連研究

関連研究について論じる．2.5.1 項と 2.5.2 項において，OCL モデルの誤り防止と，*Activity* の処理順序の検証に関する従来の研究を説明する．これらの従来技術は何れも有用な技術であるが，2.4.4 項で述べた，OCL の言語特性（文脈依存性・非決定性・不完全性）や合成モデルの複雑性の課題に対処するための実用的な手段を備えていない．本研究では課題を解決するために，抽象解釈とデータフロー解析の技術を適用する．2.5.3 項と 2.5.4 項において，本研究の土台となる抽象解釈とデータフロー解析について，技術の概要と従来研究の説明を行う．2.5.5 項では，関係データベースの表間の参照整合性を実現するための仕組みである参照制約動作の概念を説明する．

2.5.1 OCL モデルの誤り防止

従来，OCL モデルの記述と解釈に関わる誤りの発生を防ぐために，記述のパターン化，解釈のパターン化，整合性検証の技術が提案されている．

記述のパターン化

OCL モデルの記述を行う際の誤りを低減する為に，記述のパターン化が提案されている．Miliauskaitė et al. [56] は，クラス図に付加される不変条件の記述を分類して，ステレオタイプを付与する提案を行った．また Costal et al. [57] は，制約の記述パターンを UML の Profile として定義した．Wahler [55] は，複数の制約パターンを組み合わせて複合的な制約パターンを構成する方法を述べた．Ackermann et al. [75] は，事前・事後条件と不変条件の制約パターンを定義し，パターンのテンプレートを用いて OCL 記述を自動生成するツールについて述べた．また Correa et al. [76] は，解釈の誤りが起こりやすい難解な OCL 記述の形式を挙げ，リファクタリングを行うための記述パターンを提案している．表 2.4 に，Correa et al. [76] の記述パターンの例を示す．

記述パターンは，OCL 記述の誤りを低減するための経験則として有用であるが，適用範囲が限られている．また，実際のシステム開発において，記述パターンが広く普及しているとは言えない状況である．事実，Correa et al. [76] では，OCL の専門家が策定した UML の仕様書に，難解な OCL 記述が記載されている事例に言及している．このことは，一般の実践者によって作成される DC の振舞いモデルには，より高い割合で難解な表現が含まれており，記述

[*16] アジャイル型の開発プロセスは反復型の開発プロセスの派生であり，イテレーションの期間を短く設定して，顧客のフィードバックを頻繁に抽出してリリースに反映する点が特徴として挙げられる．

表 2.4. Correa et al. [76] のリファクタリングパターン例

複雑な OCL 表現	リファクタリングされた表現
`X implies (Y implies Z)`	`(X and Y) implies Z`
`X->forAll(x|x.Y->forAll(y|y.Z->forAll(z|P(z))))`	`X.Y.Z->forAll(z|P(z))`
`X->select(x|P(x))->size()>0`	`X->exists(x|P(x))`

の誤りが起こりがちであることを示唆している．

解釈のパターン化

OCL モデルの解釈を行う際の誤りを防ぐ為に，解釈のパターン化が提案されている．宣言的な OCL 記述の解釈には，2.4.2 項で述べたフレーム問題が存在する．Borgida et al. [81] が述べているように，一般にフレーム問題に対して 2 つのアプローチが取られる．一方のアプローチはフレーム仮定と呼ばれ，事後条件中で言及されないモデル要素の状態は操作を通じて変化しない，という仮定の下に事後条件を解釈する．Cabot [77]，Sendall et al. [82] が，それぞれフレーム仮定に基づく OCL 記述の解釈方法を述べている．他方のアプローチはフレーム公理と呼ばれ，モデルの記述者が操作を通じて状態が変化する（或いは変化しない）モデル要素を明示的に指定する．Kosiuczenko [83] は，フレーム公理に基づく OCL の記述方法の拡張を提案している．

また，操作の事後条件を満たす振舞いが複数存在する場合に，仕様を元にコンポーネントの操作を実装する上で，振る舞いの解釈が一意に定まらないという問題がある．この問題に対して Cabot [77] は，モデル駆動開発においてモデルからソースコードへの自動変換を実現する目的で，事後条件の宣言的な OCL 記述をパターンに基づいて命令的なアクション記述に変換する，発見的な解釈方法を提案している．しかし Cabot の提案は，OCL の論理式の文脈依存性・非決定性・不完全性を考慮しておらず，事後条件の意味を過度に限定して解釈する “overspecification”の恐れがある．

UML/OCL の整合性検証

UML/OCL モデルの性質を調べるために，整合性検証の技術が用いられている．この技術は，UML モデルと，モデルに付加された OCL 制約の間の整合性の体系的な判定を可能にする．

この技術の難しさは，検証の計算量と計算可能性にある．Berardi et al. [84] は，UML クラス図の性質に関する推論の計算量が，最悪の場合にモデル規模に対して指数関数的に増大すると報告している．また OCL は一階述語論理にオブジェクト指向拡張を施した言語であり (Beckert et al. [85])，取り得るオブジェクト状態に対して網羅的にモデルの性質を調べて完全な結果を得ることは計算不可能（undecidable）である．そこで従来，検証の自動化，OCL の表現力，検証の完全性の間のトレードオフを扱うために，以下の方法が提案されている．

定理証明技術の利用： 定理証明の技術を用いて UML/OCL モデルの検証を行う．検証ツール HOL-OCL（Brucker et al. [86]）は，OCL 記述を高階論理式にマッピングして，定理証明器 Isabelle を用いて OCL 記述の検証を行う．検証ツール KeY（Beckert et al. [87]）は，OCL 記述を一階述語論理の拡張である dynamic logic にマッピングして，UML/OCL モデルの検証を行う．この方法は，OCL の全ての表現に対応したモデルを検証することが可能であるが，証明の実行の際に利用者の介入が必要であり，検証の自動化が難しい．またツールを使いこなすために定理証明に関する詳細な知識を要する点が，一般の CBD に広く適用する上での弱点である．

局所解の探索： 限定されたオブジェクト状態の範囲の中で，OCL 制約を充足する解を探索することによって，検証を行う．検証ツール UMLtoCSP（Cabot et al. [88, 89]）は，UML モデルと OCL の不変条件・事前条件・事後条件を制約充足問題にマッピングして，制約ソルバ $\mathrm{ECL^iPS^e}$ を用いて検証を行う．検証ツール UML2Alloy（Anastasakis et al. [90]）は，UML/OCL モデルを Alloy 言語（Jackson [91]）の記述にマッピングして，Alloy 解析器（Jackson [92]）を用いて検証する．検証ツール USE（Gogolla et al. [93]）は，ツール独自の言語 ASSL: A Snapshot Sequence Language を用いて利用者が明示的に生成したオブジェクト状態に対して，OCL の不変条件を評価する．この方法では，検証の自動化が可能であるが，探索範囲の限定によって検証の完全性が失われる．すなわち探索の範囲内で反例が得られない場合には，モデルの性質について確実な結論を得ることが出来ない．また DC の場合には，データアクセス操作がインターフェースとして公開され，操作の呼び出しの度に DC のオブジェクト状態が変化し得るため，DC の仕様定義者が一律に探索範囲を定めることは難しい．

論理体系の変換： OCL 記述を計算可能な他の論理体系にマッピングして検証を行う．Queralt et al. [94, 95] は，UML クラス図と OCL の不変条件を，一階述語論理のサブセットである演繹論理にマッピングして検証する方法を提案している．この方法では，計算可能性を保証するために，モデル中で利用可能な OCL 言語要素が限定される．Queralt らの計算方法は EXPTIME 完全であり，実問題で扱われる大規模なモデルを実用的な計算コストで検証することが困難である．

2.5.2 UML Activity の処理順序の検証

従来，*Activity* の制御の流れの情報に基づいて，*Activity* 中の処理の実行順序に関する性質を検証する 2 種類の技術が提案されている．

モデル検査技術の適用

Activity を他の形式のモデルに変換した後に，モデル検査技術を用いて検証を行う．Storrle [96] は，*Activity* の制御フローの意味を，離散事象システムのモデルの一つである Petri Net にマッピングして解釈する方法を述べている．Thierry-Mieg et al. [97] は，*Activity* を Petri Net に変換して，デッドロック等の性質を検出する方法を述べている．Engels et al. [98]

は，*Activity* を変換する為の Meta モデル（DMM: Dynamic Meta Modeling）を定義し，CTL: Computation Tree Logic（Clarke [99]）を用いて soundness の性質を検証する方法を述べた．Eshuis et al. [100, 101] は，*Activity* をオートマトンに変換して，線形時相論理（LTL: Linear Temporal Logic）で記述された性質を検証する方法を述べている．Guelfi et al. [102] は，*Activity* を Promela 記述に変換して検証する方法を述べている．Abdelhalim et al. [103] は，*Activity* をプロセス代数の一種である CSP: Communicating Sequential Processes（Hoare [104]）の記述に変換して，デッドロックの有無を検証する方法を述べている．Kraemer et al. [105] は，*Activity* を時相論理の一種である cTLA: compositional Temporal Logic of Actions（Kraemer et al. [106]）の記述に変換して検証する方法を述べた．

これらの方法では，*Activity* 中の *Action* の実行順序に関する性質を検証することが出来るが，各 *Action* の入出力データの関係を考慮しないため，DC 間に跨る参照に関して，データアクセス操作の利用順序を検証する目的には適さない．

データ整合性の自動検証

Planas et al. [107, 108] は，*Activity* の制御フロー上の実行パスを解析し，データ整合性の自動検証を行う手法を提案した．この手法では，*Activity* の制御フロー上の *Action* 記述の列 <n,act>+ を解析する．n はアクション act の実行回数であり，+ は繰り返しを表す．データ整合性を破壊する可能性がある *Action* の種別毎に，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．例えば必須属性 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 et al. [107, 108] 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，*ObjectFlow* や，変数を介したデータの流れを考慮しておらず，複数の DC のデータ間の関係を生み出す，*Activity* 中のデータ伝搬を扱うことが出来ない．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．

2.5.3　抽象解釈

抽象解釈は，プログラムの性質を近似的に解析するための理論的な枠組みであり，Cousot et al. [109] で提案された．手続型プログラム言語に関する研究に加えて，関数型プログラムや論理プログラムなどの宣言型言語に関する研究が数多く存在する（堀内 [110]）．抽象解釈の基本的な特徴は，プログラムが扱う型の領域（具体領域）を抽象化した抽象領域を定義し，抽象領域上でプログラムの意味を近似的に解釈する点である．

抽象解釈の概念を図 2.18 に示す．具体領域 D の各要素は，抽象化写像 $\alpha : D \rightarrow \underline{D}$ によって，抽象領域 $\underline{D}$ の要素に対応付けられる．一方，抽象領域の要素は具体化対応 $\gamma = \alpha^{-1}$ によって具体領域の部分集合に対応付けられる．具体領域 D 上では解釈 I に基づいてプログラ

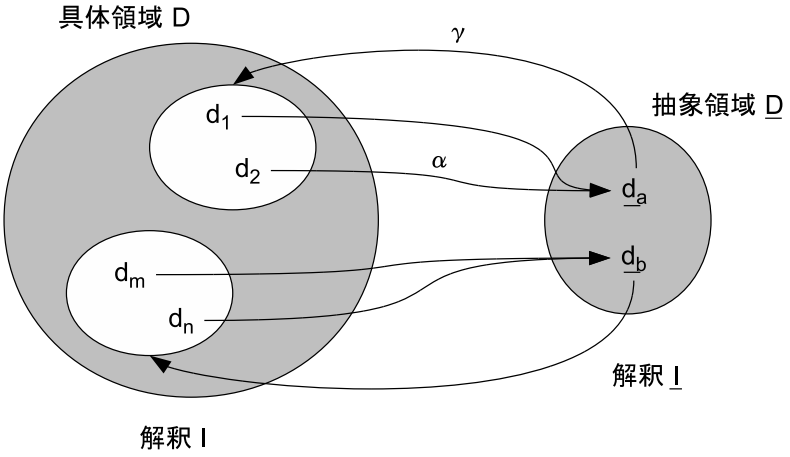


図 2.18. 抽象解釈の概念

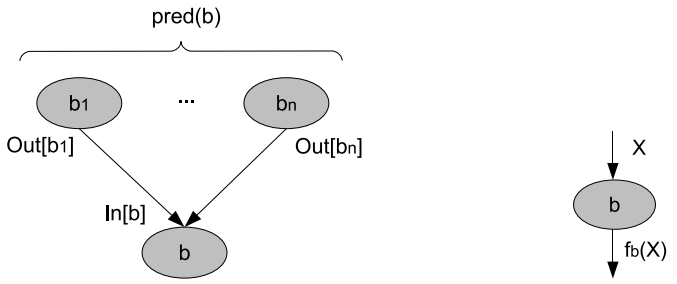
ムが実行されるのに対して，抽象領域 $\underline{D}$ 上では，近似的な解釈 $\underline{I}$ に基づいて不動点を計算し，プログラムの意味の近似的な解釈を行う．

抽象解釈は，近似によって，実際の実行よりも少ない計算でプログラムの性質を得ることが出来る利点がある．一方，過度な近似を行った場合に，プログラムの情報が失われて，必要な性質が解析できない恐れがある．また，一般に近似計算の結果には，実際の実行では生じないノイズ情報が含まれるが，実際の実行に対応する情報が漏れなく含まれる，安全な近似であることが望ましい．抽象解釈を適用する際には，求めるプログラムの性質に応じて，近似の計算時間，精度，安全性を考慮して $\underline{D}$，α，γ，$\underline{I}$ を適切に定める必要がある．

UML/OCL モデルに対する抽象解釈の研究は数少ない．Baruzzo et al. [111] は，クラス図と OCL 制約記述に対するシーケンス図の整合性検証の計算量を削減する目的で，抽象解釈の適用を提案している．しかし Baruzzo et al. [111] には，抽象化の具体的な方法が示されていない．Cabot et al. [112] は，UML クラス図に関する OCL の不変条件を破壊する可能性がある，モデル要素の変更アクションを特定する手法を提案している．Cabot et al. [112] の手法は，OCL の抽象構文木の各ノードに，破壊の有無を表す 3 通りの抽象記号（“+”，“–”，“und”）を割り当てて，OCL 記述の意味を解釈する点が，抽象解釈と類似している．ただし Cabot et al. [112] では，抽象解釈との関係が明示的に述べられていない．

2.5.4 データフロー解析

本研究では，データフロー解析（DFA: Data-flow analysis）の技術を用いて，PC の振る舞いを表す *Activity* を静的に解析する．DFA は，着目する情報が制御フローグラフ（CFG : Control Flow Graph）の各点で取りうる値集合を求める解析技法であり，Kindall [113] によって最初に提案された．これまでに，コンパイラの最適化やデータ異常の検出などを主なアプリケーションとして，プログラムを対象にした数多くの研究が行われてきた．DFA は，ソフトウェアの動作時の履歴情報を利用する動的 DFA と，ソフトウェアを動作させずに解析を



(a) 直前ブロックからの伝搬　　(b) 伝達関数

図 2.19. 前方解析

行う静的 DFA に大別されるが，本論文では後者の静的 DFA を扱う．

CFG は，プログラムやモデル中の分岐を含まない基本ブロックをノードとして，基本ブロック間の制御の流れを有向辺で表した有向グラフである．静的 DFA では，CFG の各ノードについて，情報の伝搬を表すデータフロー方程式を定義し，ノード上で取りうる情報の値集合を，方程式に基づき繰り返し計算して不動点として求める．データフロー方程式は，対象ブロックの前方からの伝搬を表現する前方解析（forward analysis）と，対象ブロックの後方への伝搬を表現する後方解析（backward analysis）に分類される．

解析の方法には，条件分岐の内容に従って経路を考慮する経路依存（path-sensitive）の方法と，経路を考慮しない経路非依存（path-insensitive）の方法がある．手続間に跨って解析を行う場合には，手続内のフローの情報を考慮するフロー依存（flow-sensitive）の方法と，手続内のフローを考慮しないフロー非依存（flow-insensitive）の方法がある．さらに，手続の呼び出し箇所の文脈を考慮する文脈依存（context-sensitive）の方法と，文脈を考慮しない文脈非依存（context-insensitive）の方法がある．これらの方法では，解析の複雑さと精度がトレードオフの関係にある．経路非依存，フロー非依存，文脈非依存の方法では，経路依存，フロー依存，文脈依存の方法に比べて，それぞれ計算を単純化することが出来るが，一般に解析の精度が低下する．

前方解析のケースを例にして，データフロー方程式の導出方法を説明する．基本ブロックの集合 B，情報の値集合 D に対して，伝搬に関わる以下の変数を定義する．

$In[b] \subseteq D$	$b \in B$ への入力
$Out[b] \subseteq D$	b の出力
$pred : B \to \mathfrak{B}(B)$	直前ブロックの集合を得る関数
$f_b : D \to D$	b における伝達関数

このとき，ブロック b について次の 2 つの等式を考えることができる[*17]．

$$In[b] = \bigcup_{b_i \in pred(b)} Out[b_i],$$
$$Out[b] = f_b(In[b]).$$

第 1 の等式は，図 2.19(a) に示すように，b とその直前ブロック $b_i(i = 1, .., n)$ の間の情報の伝搬を表している[*18]．b の入力は，経路に依らず，全ての直前ブロックの出力の総和として定義されており，この等式に基づく解析は経路非依存である．第 2 の等式は，図 2.19(b) に示すように，b における情報の伝達を表している．等式の $In[b]$ と f_b を，着目する情報に応じて具体的に設定することによって，特定用途向けのデータフロー方程式が得られる．

DFA の主な用途として，CFG 中の変数や式の定義の解析が挙げられる．代表的な定義の解析として，対象ブロックに到達する可能性がある変数や式の定義箇所を求める到達定義解析や，対象ブロックで定義された変数が使用される可能性のある箇所を求める定義使用解析などがある． 定義の対象が構造を有するオブジェクトの場合には，オブジェクトの属性毎に伝搬を調べる属性依存（field sensitive）の解析が必要となる．特に 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に，Wilson et al. [114]，Yong et al. [115]，Pearce et al. [116] が，構造体の属性のオフセットを利用して属性依存の解析を行う方法を提案している．

近年では, モデリング言語を対象とした定義の解析の研究も行われている．Sadiq et al. [117] はワークフローモデルのデータ不整合のパターンを挙げ，不整合の検出に DFA を適用することを提案した．UML Activity の定義の DFA に関しては，Meda et al. [118] がデータフロー方程式を用いた解析方法を，Storrle [119] がカラーペトリネットへのマッピングによる解析方法を，それぞれ提案している．また Waheed et al. [120] は，$Action$ 記述を含む UML の状態遷移モデルについて，変数の定義使用解析の方法を提案した．しかしこれらの方法は，何れもオブジェクトの属性値の伝搬を考慮しない，属性非依存（field insensitive）の解析方法である．

DFA は，CFG 中の順序制約の検証にも適用されている．Olender et al. [121] は，プログラムの手続内のイベント生起の順序を静的に検証する手法を提案した．さらに Olender et al. [122] は，手法の適用範囲を手続間に拡張した．Olender et al. [121, 122] の手法では，順序制約を有限オートマトン M として表現し，プログラムを表現する CFG の各ノード上で M が取り得る状態を，DFA によって解析する．

DFA とモデル検査の関係について述べる．Schmidt [123] は，或る種のデータフロー方程式を μ 計算を用いて表現し，それらの方程式の DFA が，データ抽象化を施した CFG に対するモデル検査と見なすことが出来ると述べた．Steffen [124, 125] は，DFA のアルゴリズムをプログラムに対する様相論理として表現し，モデル検査器を用いて DFA を実行するアプローチを述べている．

[*17] CGF の制御の入口となるブロック $entry$ に関しては．$In[entry]$ が別途与えられる．

[*18] 後方解析の場合には，第 1 の等式の代わりに，後続ブロックを求める関数 $succ : B \to \mathfrak{B}(B)$ を用いて，b と後続ブロックの間の情報の伝搬を表現する．

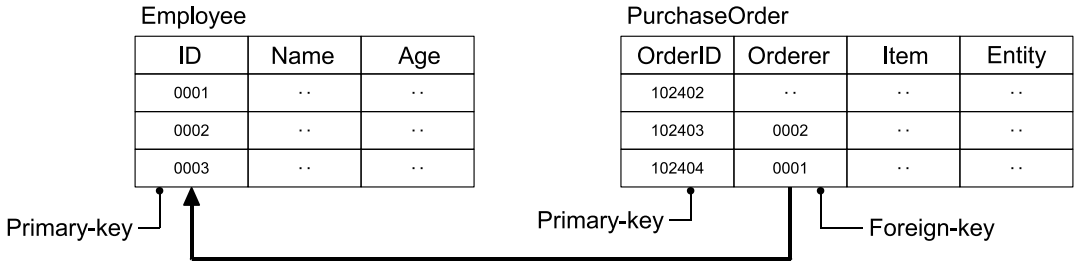


図 2.20. 被参照表と参照表の例

2.5.5　参照制約動作

代表的なデータ管理方法の 1 つである関係データベースでは，参照表に，被参照表の主キーを参照する外部キーを定義することによって，表間の参照関係を表す．図 2.20 に被参照表と参照表の例を示す．この例では，参照表 `PurchaseOrder` の外部キー `Orderer` が，被参照表 `Employee` の主キー `ID` を参照している．

被参照表に対するレコードの削除や主キーの更新，参照表に対する外部キーの更新の際に，表間の参照の整合性を維持するために，被参照表と参照表の内容を同期させる必要がある．データベース言語標準の SQL2003 では，表間の参照整合性を強制的に適用するための仕組みとして，次の 5 種類の参照制約動作（Referential action）を規定している．

CASCADE： 被参照表のレコードの削除，主キー値の更新に連鎖して，参照表の対応するレコードの削除，（主キー値と同じ値に）外部キー値の更新を行う．また参照表の外部キー値の更新において，更新後の外部キー値を主キーに持つレコードが被参照表に存在するときのみ，外部キー値の更新を行う．

RESTRICT： 被参照表のレコードの削除，主キー値の更新において，該レコードが参照されていないときのみ，レコードの削除，主キー値の更新を行う．また参照表の外部キー値の更新において，CASCADE と同じ動作を行う．

NO ACTION： RESTRICT と同様の動作を行うが，レコードの削除や更新の事後に参照整合性チェックを実施して，整合性が破壊される場合にロールバックを行う点が，RESTRICT と異なる．

SET NULL： 被参照表のレコードの削除，主キー値の更新に際して，参照表の対応するレコードの外部キー値を NULL に更新する．

SET DEFAULT： 被参照表のレコードの削除，主キー値の更新に際して，参照表の対応するレコードの外部キー値を既定値に更新する．

参照制約動作は，参照表の定義中に，外部キーに関する制約の一部として記述される．図 2.20 の参照関係について，参照制約動作の記述例を図 2.21 に示す．17 行目の記述によって，レコードの更新の際に CASCADE が，レコードの削除の際に RESTRICT が，それぞれデー

```
1   CREATE TABLE Employee (
2      ID              VARCHAR(4) NOT NULL,
3      Name            VARCHAR(20) NOT NULL,
4      Age             INTEGER NOT NULL,
5      CONSTRAINT id_pk PRIMARY KEY(ID),
6      CONSTRAINT age_value CHECK (Age > 17) )
7
8   CREATE TABLE PurchaseOrder (
9      OrderID         INTEGER NOT NULL,
10     Orderer         VARCHAR(4) NOT NULL,
11     Item            VARCHAR(40) NOT NULL,
12     Entity          INTEGER NOT NULL,
13     CONSTRAINT orderid_pk PRIMARY KEY(OrderID),
14     CONSTRAINT entity_value CHECK (Entity > 0),
15     CONSTRAINT orderer_fk FOREIGN KEY(Orderer)
16        REFERENCES Employee(ID)
17        ON UPDATE CASCADE ON DELETE RESTRICT )
```

図 2.21. 参照制約動作の記述例

タベース管理システムによって実行され，表間の参照整合性が保たれる．

2.3.2 項で定義した DC 間の参照 $r = (cls_{src}, prop_{src}, cls_{dst}, prop_{dst}) \in REF$ は，属性値を介した間接的な参照関係である点が，関係データベースの表間の参照に類似している．cls_{src} と cls_{dst} は，参照表と被参照表に，$prop_{src}$ と $prop_{dst}$ は，外部キーと主キーに，それぞれ対応づけられる．ただし，DC 間の参照と，関係データベースの表間の参照には，明確な違いが存在する．関係データベースにおける被参照表と参照表が，同一のデータベース管理システムで管理されるのに対して，cls_{src} と cls_{dst} は異なる DC によって独立に維持管理される．

PC と DC の合成モデルの振舞いにおいて，参照制約動作と同様のアプローチによって，DC 間の参照の整合性を実現するケースがある．例えば，図 2.15 中の参照 $\mathtt{r_1}$ の参照整合性を維持するための合成モデルの振舞いは，参照元への値の代入（手順 E）に先立って，代入値をキー属性に持つ参照先のオブジェクトの存在を照合する（手順 B）．この振舞いは，参照表の外部キー値の更新を行う際の，CASCADE あるいは RESTRICT の動作と類似している．

一方，図 2.16 中の参照 $\mathtt{r_2}$ の参照整合性を維持するための合成モデルの振舞いは，参照先の属性値を取得（手順 F）して，その値を参照元へ代入する（手順 J）．これは参照制約動作ではなく，データベース・アプリケーションのコード中の一連の処理シーケンスによって，関係データベースの表間の参照整合性を実現するケースに類似している．

第3章

OCL記述の解析

本章では，DC のデータアクセスの整合性（整合性 1）の判定を支援するための OCL 記述の解析手法の説明を行う．2.4.4 項で述べたように，整合性 1 の判定を人手で行う必要があるが，OCL の言語特性によって判定の誤りが生じる恐れがある．本解析手法は，DC の振舞いモデルを静的に解析して，整合性 1 の判定を行うためのアクセス情報を提供する．本手法の解析を自動実行して，その出力情報を用いて判定を行うことにより，人手による整合性判定の正確性と効率の向上が可能になる．

3.1 節で解析の概要を述べる．3.2–3.7 節において，解析の 6 つの手順（抽象解釈，起源の解析，データ流向の特定，確定性の判定，値の対応付け，含意判定）を説明する．3.8 節で，手法の適用を支援するツールの説明を行う．

3.1 概要

整合性 1 とは，DC のデータへのアクセスに関する，DC の振舞いモデルの表現と，記述者の意図や利用者の解釈が無矛盾，すなわち一致することである．DC の振舞いモデルの表現と，それに対するモデル記述者・利用者の認識の関係を，図 3.1 に模式的に示す．

2.4.1 節で述べたとおり，モデル記述者の意図や利用者の解釈そのものは形式的に表現されないため，モデルの表現全体が示す範囲と，それに対する人の認識の範囲の一致を判定するた

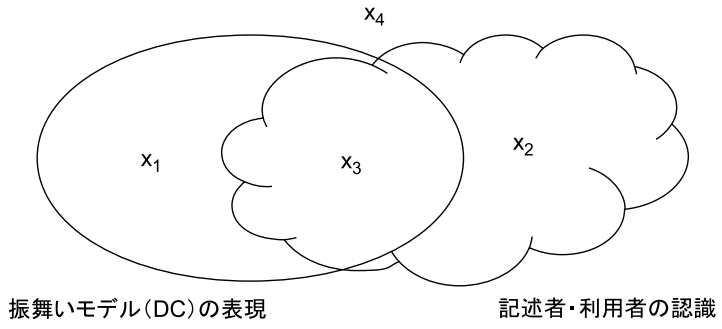


図 3.1. モデルの表現と人の認識

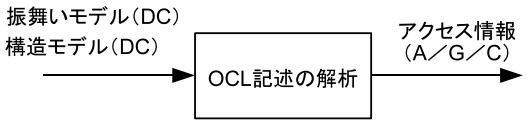


図 3.2. OCL 記述の解析の入出力

めの条件を明確に定義することはできない．しかし，ある特定の性質 x について，モデルの表現と人の認識の一致を確認することによって，x に関する両者の整合を判定することは可能である．次の (1)–(4) の中で，(1) と (2) では不整合が存在し，(3) と (4) では整合がとれている．

(1) モデルの表現は性質 x を示すが，人は x を認識しない（$\mathrm{x} = \mathrm{x}_1$ のケース）．
(2) モデルの表現は性質 x を示さないが，人は x を認識する（$\mathrm{x} = \mathrm{x}_2$ のケース）．
(3) モデルの表現は性質 x を示し，人も x を認識する（$\mathrm{x} = \mathrm{x}_3$ のケース）．
(4) モデルの表現は性質 x を示さず，人も x を認識しない（$\mathrm{x} = \mathrm{x}_4$ のケース）．

OCL 記述の解析は，DC の振舞いモデルの要点であるデータアクセスに関する性質を扱う．

3.1.1 解析の入出力

本解析は，DC の構造モデルと振舞いモデルを入力として，アクセス情報 AC の構成要素である代入情報 A, 取得情報 G, 照合情報 C を出力する（図 3.2）．$AC = (A, G, D, C)$ はデータアクセスの性質を示す情報であり，$A, G, D \subseteq Operation \times Class \times Property \times Parameter \times (Property \cup \mathrm{NULL})$, $C \subseteq Operation \times Class \times Property \times Parameter \times (Property \cup \mathrm{NULL}) \times Parameter \times (Property \cup \mathrm{NULL})$ 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．cls と $prop_{cls}$ の組を代入先情報，$param$ と $prop_{param}$ の組を代入値情報と呼ぶ．
- 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．cls と $prop_{cls}$ の組を取得先情報，$param$ と $prop_{param}$ の組を取得値情報と呼ぶ．
- 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．cls と $prop_{cls}$ の組を削除先情報，$param$ と $prop_{param}$ の組を削除指定情報と呼ぶ．
- 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[*1]．cls と $prop_{cls}$ の組を照合先情報，$param_{in}$ と $prop_{in}$ の組を照合値情報，$param_{out}$ と $prop_{out}$ の組を照合出力情報と呼ぶ．

操作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．DC の各操作の振舞いモデルには，0 個以上の複数の A, G, D, C の要素が対応する．

2.3.3 項で説明した AccessConfig::getAccessibleWS の振舞いモデル（図 2.8）の場合，$G = \{(\texttt{getAccessibleWS}, \texttt{WSDef}, \texttt{id}, \texttt{result}, \texttt{id}), (\texttt{getAccessibleWS}, \texttt{WSImp}, \texttt{version}, \texttt{result}, \texttt{version})\}$，$C = \{(\texttt{getAccessibleWS}, \texttt{Project}, \texttt{id}, \texttt{projectID}, \texttt{NULL}, \texttt{result}, \texttt{NULL})\}$ である．データの代入と取得のアクセスは図 2.8 に記述されていないため $A = D = \phi$ である．WSStatusMgr::addAccessor の振舞いモデル（図 2.9）の場合は，$A = \{(\texttt{addAccessor}, \texttt{WSInfo}, \texttt{wsDefID}, \texttt{wsDefID}, \texttt{NULL}), (\texttt{addAccessor}, \texttt{WSInfo}, \texttt{wsImpVersion}, \texttt{wsImpVersion}, \texttt{NULL}), (\texttt{addAccessor}, \texttt{Accessor}, \texttt{projectID}, \texttt{projectID}, \texttt{NULL})\}$，$G = D = C = \phi$ である．

PC の振舞いモデルの *Activity* に記述される，DC のデータアクセス操作の呼び出しは，対象操作の振舞いモデルから導出される 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 2.15 の番号 3 の *CallOperationAction* ノードは，照合アクセス $(\texttt{getAccessibleWS}, \texttt{Project}, \texttt{id}, \texttt{projectID}, \texttt{NULL}, \texttt{result}, \texttt{NULL}) \in C$ の実行を表している．2.4.2 項と 2.4.3 項で説明したとおり，PC の振舞いモデルにおいてデータアクセスを適切な手順で実行することによって，DC 間の参照の整合性が保たれる．そのような *Activity* を記述するためには，DC の振舞いモデルが示すアクセス情報 AC と，人の認識が一致することが前提となる．

2.5.5 項で述べたように，PC の振舞いモデルにおけるデータアクセスの手順は，関係データベースの表間の参照の整合性を保つための参照制約動作や，データベース・アプリケーションのコードの処理シーケンスに類似している．参照制約動作のトリガとアクションとして削除と更新（代入）のアクセスが行われ，表中のレコードの存在を確認するために照合のアクセスが行われる．また，データベース・アプリケーションのコードでは，参照先データの取得のアクセスが行われる．これらのアクセスは，アクセス情報 AC の構成要素 A，G，D，C のいずれかに対応する．従って，DC 間の参照整合性を確保する観点において，アクセス情報 AC は整合性 1 の判定に必要な情報を備えている．

削除情報 D の抽出は，本解析手法の範囲外である．2.4.4 項で述べたように，OCL にはオブジェクトの削除を明示的に表現する演算子が存在しない．そのため，本論文で提案する解析手法では削除情報 D を抽出することができない．削除情報 D の抽出方法は，今後の研究課題である．

[*1] 本論文では実問題において主流である，単一の出力パラメータにより照合結果を伝える照合アクセスを扱う．

2.4.4 項で述べたとおり，宣言的な仕様記述の解釈において，事後条件に明示的に記述されないモデル要素の状態が確定しない，というフレーム問題が存在する．本手法では UML Components（Cheesman et al. [47]）と同様に，事後条件に記述されないモデル要素の状態は操作を通じて変化しない，というフレーム仮定の下に事後条件を解釈することによって，フレーム問題を解消する．この方法は，フレーム公理に基づく言語拡張の方法（Kosiuczenko [83]）に比べて，OCL の言語標準に従うモデルを扱うことができるメリットを持つ．すなわち，UML Components などの方法論に従って定義されたモデルを広く扱うことができる．Kosiuczenko [83] が指摘しているように，一般にフレーム仮定を用いた場合には，モデル要素間の継承関係の解釈に矛盾が生じうるが，本手法が扱う構造モデルは継承関係を含まないため，矛盾の恐れはない．

本手法は，DC の振舞いモデルの中で，データアクセス操作の事後条件を対象として解析を行う．代入情報，取得情報，照合情報は，操作内容を規定する情報として事後条件に記述されるはずである．またフレーム仮定より，事後条件には操作の実行による全ての効果が記述されるため，DC の振舞いモデルが代入・取得・照合のデータアクセスを表現する場合には，データアクセス操作の事後条件にその記述が含まれるはずである．従って，操作の事後条件を対象に静的解析を行えば良い．

3.1.2 技術課題と解決アプローチ

アクセス情報 AC を導出する上での技術課題と，本解析手法で採用する課題解決アプローチを表 3.1 にまとめる．

DC の振舞いモデルはオブジェクト状態に対する制約として記述されるが，制約を満たす状態は無限に存在しうる．そのため，とりうる状態を完全に扱うことができる静的解析の方法が必要である．しかし，2.5.1 項で述べたとおり，UML/OCL モデルの検証において，検証の自動化と完全性，モデルの表現力の 3 つの間にトレードオフが存在する（技術課題 1）．本手法では，2.5.3 項で述べた抽象解釈技術を利用して，近似的な解析を行うことによって，技術課題 1 の解決に取り組む．

2.5.1 項で説明した既存の UML/OCL モデルの検証手法と本手法の比較を図 3.2 に示す[*2]．

表 3.1. OCL 記述の解析：課題解決のアプローチ

	技術課題	解決のアプローチ
1	自動化，完全性，モデル表現力のトレードオフ	抽象解釈技術
2	文脈依存性・非決定性・不完全性	OCL の意味論に従う抽象解釈規則
3	代入・取得・照合の識別	起源の解析，データ流向の特定
4	照合情報の解析の計算量	抽象解釈の応用による計算の効率化

[*2] 抽象解釈が近似的な解析であることから，比較の観点に正確性・網羅性を加えている．局所解の探索による手法では，探索の範囲内で反例が得られない場合の計算結果が，正確性・網羅性に欠ける可能性がある．

表 3.2. OCL 解析方法の比較

	自動化	完全性	モデルの表現力	正確性・網羅性
定理証明	×	○	○	○
局所解の探索	○	×	○	△
論理体系の変換	×	○	△	○
本手法	○	△	△	△

表中の “△” は，手法の能力に制限があることを示す．本手法では，抽象解釈の近似計算によって，オブジェクト状態を完全に扱いながら，解析の自動化を実現する．ただし，一部の OCL 演算子を含む OCL 記述については，近似の安全性が保証されず，オブジェクト状態の扱いが完全ではない．そして，近似の精度不足により，解析の正確性・網羅性が失われる場合がある．また，抽象化の際に *Collection* 型の要素の順序の情報を捨象するため，*Sequence* と *OrderedSet* の演算を含むモデルを扱うことが出来ない．

2.4.4 項で述べたように，OCL の論理式は文脈依存性・非決定性・不完全性の性質を有する（技術課題 2）．本手法では，OCL の意味論に従う抽象領域上の解釈規則を定義することによって，これらの OCL の性質を厳密に扱う．

2.4.4 項で述べたとおり，OCL にはデータに対する代入，照合，取得のアクセスを表現する専用の演算子が存在しないため，それらのアクセスの識別が難しいという問題がある（技術課題 3）．本手法では，OCL 表現ノードの具体的な値が由来するモデル要素（起源）を解析し，OCL 表現ノード間のデータの流向を特定することによって，代入，照合，取得のアクセスを識別する．

さらに，照合情報 C を求める際に，照合の出力と結果の対応を得るために，照合を表す OCL 表現ノードと，照合結果を出力する OCL 表現ノードの値の組を求める必要がある．この値の組は，抽象構文木中のノードの意味の解釈の組合せで決まるが，そのような組合せの数は，ノード数に対して指数関数的に増大する（技術課題 4）．実問題の典型的な事後条件のノード数は数十から数百であり，組合せを総当たりで計算することは困難であるため，効率的な計算方法が必要である．本手法では，抽象解釈技術を応用して，OCL 表現ノードの値の組を与える解釈の組合せの有無を求めることにより計算を効率化し，技術課題 4 を解決する．

3.1.3　解析手順

解析手法は，代入・取得・照合を表す OCL 表現ノードを求める Step 1 と，照合結果の出力を表す OCL 表現ノードを求める Step 2 の 2 つのステップからなる（図 3.3）．Step 1 で，OCL の意味論を厳密に扱う抽象解釈の方法を導入して技術課題 1，2 を解決し，起源の解析とデータ流向の特定を行うことによって，代入・取得・照合のアクセスを識別して技術課題 3 を解決する．さらに Step 2 では，抽象解釈の技術を応用して，照合の出力値と照合結果の値の組を与える各ノードの解釈の組合せの存在の有無を求めることにより，計算を効率化し，技術課題 4 を解決する．

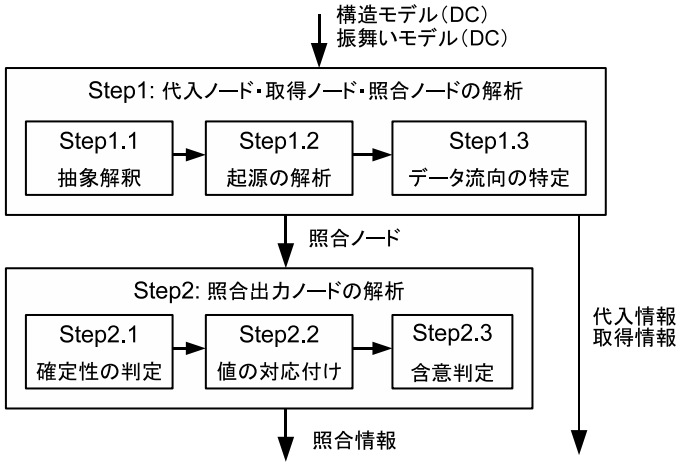


図 3.3. OCL 記述の解析手順

Step 1：**代入ノード・取得ノード・照合ノードの解析**

2.3.1 項で述べたように，OCL 事後条件の抽象構文木は，OCL 表現ノードから構成される．本手法では，抽象構文木上のデータアクセスを表す OCL 表現ノードを以下のように定義する．

- 代入ノード：入力パラメータからデータオブジェクトの属性への値の代入を表す．
- 取得ノード：出力パラメータを介したデータオブジェクトの属性値の取得を表す．
- 照合ノード：入力パラメータ値とデータオブジェクトの属性値の間の照合を表す．

代入ノードと取得ノードは，データオブジェクトの属性とパラメータの間の値の割り当てを意味する．そのような値の割り当ては，割り当て後に属性とパラメータの間に成立する関係として宣言的に表現されるはずである．そして，OCL ではそのような関係を表現可能な形式は，演算 “=”を用いた等価関係，もしくは演算 “`includes`”，“`includeAll`”を用いた包含関係に限定される．

照合ノードは，データオブジェクトの属性とパラメータの間の値の等価比較を意味する．代入ノード，取得ノードと同様に，OCL で等価比較を表現可能な形式は，“=”を用いた等価関係，もしくは “`includes`”，“`includeAll`”を用いた包含関係に限られる．

以上の考察から，本手法では代入ノード，取得ノード，照合ノードがいずれも次の形式を持つことを仮定する．

代入・取得・照合ノードの形式：演算子 “=”を用いた等価関係，または演算子 “`includes`” 或いは “`includeAll`”を用いた *Collection* の包含関係．

上記の形式で表現される代入ノードの，等価演算子あるいは包含演算子の左右のオペランドが，代入先情報と代入値情報を表す．同様に，取得ノードの場合には，左右のオペランドが取得先情報と取得値情報を表す．そして照合ノードの場合には，左右のオペランドが照合値情報と照合先情報を表し，照合ノードの値が照合結果を表す．ただし両オペランドの順序は不同で

あり，それらの形式に関する仮定を設けない．

Step 1 では，代入ノード，取得ノード，照合ノードの解析を行う．代入ノードから代入情報を，取得ノードから取得情報を，それぞれ求めて出力する．また照合ノードは Step 2 の入力として用いる．

代入・取得・照合ノードの形式は，OCL 表現ノード E が代入，取得，照合を表すための形式的な制約であり，実際に E が代入，取得，取得を表すためには，さらに意味的な制約が必要である．本手法では，代入の意味的な制約として，次の条件 1_{AG}，条件 2，条件 3_A を，取得の意味的な制約として，条件 1_{AG}，条件 2，条件 3_G を，照合の意味的な制約として，条件 1_C，条件 2，条件 3_C を，それぞれ課する．

条件 $\mathbf{1}_{AG}$： E の値が `true` となる，事後条件を満たすオブジェクト状態が存在する．

条件 $\mathbf{1}_C$： E の値が `true` または `false` となる，事後条件を満たすオブジェクト状態が存在する．

条件 2： E の一方のオペランドに操作パラメータまたはその属性が，他方のオペランドにデータモデル中の *Class* の属性値が含まれる．

条件 $\mathbf{3}_A$： 条件 2 の操作パラメータから，*Class* の属性へのデータの流れが存在する．

条件 $\mathbf{3}_G$： 条件 2 の *Class* の属性から，操作パラメータへのデータの流れが存在する．

条件 $\mathbf{3}_C$： 条件 2 の操作パラメータと *Class* の属性は，操作の実行を通じて不変であり，両者の間にデータの流れは存在しない．

条件 1_{AG} は代入と取得が実際に発生するための条件であり，条件 1_C は照合が実際に行われる状態が存在するための条件である．条件 2 は代入ノード，取得ノード，照合ノードの定義に由来する．条件 3_A と条件 3_G は，それぞれ代入と取得の記述が，値の割り当て方向を表現するために満たすべき条件であり，条件 3_C は，フレーム仮定の下で，副作用を伴わない照合の記述が満たすべき条件である[*3]．

本手法は，これらの条件を満たす代入ノード，取得ノード，照合ノードを，次の 3 つのサブステップによって求める．Step 1.1 は条件 1_{AG}，条件 1_C に，Step 1.1 は条件 2 に，Step 1.3 は条件 3_A，条件 3_G，条件 3_C に，それぞれ対応している．

Step 1.1： 抽象解釈を行い，代入・取得・照合ノードの形式を持つ OCL 表現ノードの中で，条件 1_{AG} を満たすものを代入ノード，取得ノードの候補として，条件 1_C を満たすものを照合ノードの候補として，それぞれ求める．

Step 1.2： Step 1.1 で求めた各候補に対して，オペランドの値が由来するモデル要素（起源）の解析を行い，条件 2 を満たすものをスクリーニングする．

Step 1.3： Step 1.2 で求めた代入ノード，取得ノードの候補のデータ流向を特定し，条件 3_A を満たすものを代入ノードとして，条件 3_G を満たすものを取得ノードとして，

[*3] 本手法では照合の記述として，操作の実行前と実行後で *Class* の属性値が変化しないこと，実行後の属性値と入力パラメータを照合すること，の 2 段階に分割する冗長な記述を許容しない．

それぞれ決定する．代入ノードについて，Step 1.2 で求めた左右のオペランドの起源の情報から，代入値情報と代入先情報を求めて出力する．同様に取得ノードについて，取得値情報と取得先情報を求めて出力する．また，Step 1.2 で求めた照合ノードの候補のデータ流向を特定し，条件 3_C を満たすものを照合ノードとして決定する．

3.2–3.4 節で，Step 1.1–1.3 の各サブステップの詳細な内容を説明する．

Step 2：**照合出力ノードの解析**

本手法では，照合ノードに対応する照合出力ノードを以下のように定義する．

- 照合出力ノード：照合ノードにおける，入力パラメータ値とデータオブジェクトの属性値の間の照合の結果を表す．

Step 2 では，Step 1 で求めた照合ノードに対応する照合出力ノードの解析を行う．照合ノードと照合出力ノードから，照合情報を求めて出力する．

照合出力ノードの満たすべき条件を以下に示す．

条件 4： 常に確定した値を持つ[*4]．

条件 5： 照合出力ノードと照合ノードの取り得る値の組は事後条件を満たす．

条件 6： 照合出力ノードの値から，照合結果が真となることが演繹的に決定される．

条件 6 の代わりに，照合出力ノードの値と照合結果の間に，他の条件[*5] を課すことも考えられる．しかし，照合操作は照合データの存在を確認する為に用いる操作であるため，条件 6 が妥当である．

本手法では，これらの条件を満たす照合出力ノードを，次の 3 つのサブステップによって求める．各サブステップは，それぞれ条件 4–6 に対応している．

Step 2.1： 出力パラメータまたはその属性を表す OCL 表現ノードの中で，常に確定値を持つノードを，照合出力ノードの候補としてスクリーニングする．

Step 2.2： Step 2.1 で求めた照合出力ノード候補と，Step 1 で求めた照合ノードに対して，事後条件の制約を充足する値の対応を求める．

Step 2.3： Step 2.2 で求めた値の対応から，照合出力ノードの値と，照合結果が真であることの含意判定を行い，条件 6 を満たす照合出力ノードを決定する．そして，照合ノードについて，Step 1.2 で算出したオペランドの起源の情報から，照合先情報と照合値情報を求めて出力する．同様に，照合出力ノードの起源の情報から照合出力情報を求めて出力する．

3.5–3.7 節で，Step 2.1–2.3 の各サブステップの詳細な内容を説明する．Step 2.1 の確定性の判定と，Step 2.2 の値の対応付けは，Step 1.1 の抽象解釈の解析技術に基づいており，抽象

[*4] 不定値から，照合結果を特定することは出来ないため．

[*5] たとえば，照合出力ノードの値から，照合結果が偽となることが演繹的に決定される等の条件．

表 3.3. OCL 表現の値と型

値	型
PrimitiveValue	*PrimitiveType, Enumeration*
TupleValue	*DataType*
ObjectValue	*Class*
CollectionValue (SetTypeValue,BagTypeValue)	*Collection(Set,Bag)*
OclVoidValue	All types

解釈の解析手続を呼び出し，その結果を利用する．

3.2　抽象解釈

本節では，Step 1.1 で条件 1_{AG} と条件 1_{C} の判定を行うための抽象解釈の解析方法を説明する．

2.3.1 項で説明したように，OCL 抽象構文木を構成する各 OCL 表現ノードは単一の型を持ち，オブジェクト状態に対して構文木を評価することにより，ノードの値が決定される．本手法で扱う OCL 表現ノードの型と値の対応を表 3.3 に示す．表 3.3 中の対応は，OCL 言語仕様における定義（表 2.3）のサブセットであり，*Collection* の中では要素の順序を持たない *Set*, *Bag* を解析の対象とする[*6]．また *OclVoidValue* は OCL の不完全性により生じる未定義の値を表す[*7]．

表 3.3 に示した OCL 表現の値を具体値，その集合を具体領域と呼ぶ．具体領域には *Integer* や *Class* のインスタンスなど無限個の要素が含まれる．従って具体領域上で各 OCL 表現ノードがとり得る具体値を求めるためには，無限個の要素の組み合わせを扱わなければならない．そこで具体領域の部分集合に対応する抽象値を定義し，有限の抽象値を要素とする抽象領域上で近似的な意味の解釈（抽象解釈）を行う．

3.2.1　抽象領域

具体領域 D に対して，抽象領域 $\underline{D}$ ={T,F,U} を定義する．具体領域の要素 d は，抽象化写像 $\alpha : D \rightarrow \underline{D}$ によって抽象領域の要素に対応付けられる．

$$\alpha(d) = \begin{cases} \mathrm{T} & \text{if } d \text{ が } \texttt{true}(\mathit{Boolean}),\ \text{非 } 0(\mathit{Integer/Real}),\ \mathit{OclVoidValue} \text{ 以外の任意の値 } (\mathit{DataType/Class/String/Enumeration}),\ \text{非空 } (\mathit{Collection}) \text{ の何れか} \\ \mathrm{F} & \text{if } d \text{ が } \texttt{false}(\mathit{Boolean}),\ 0(\mathit{Integer/Real}),\ \text{空 } (\mathit{Collection}) \text{ の何れか} \\ \mathrm{U} & \text{if } d \text{ が } \mathit{OclVoidValue} \end{cases}$$

[*6] 要素の順序を持つ *Sequence* と *OrderedSet* の解析方法は今後の研究課題である．

[*7] OCL 言語仕様では，0 除算等により発生する例外も *OclVoidValue* に含まれるが，本論文では事後条件中で例外を扱わない．

表 3.4. *Boolean* 型の論理演算ノードの解釈

	and		or		xor		not	implies	
T	T	T	T/U	U/T	T/F/T	T/T/F	F	T/F	T/U
F	F/U	U/F	F	F	F	F	T	T	F

一方，抽象領域の要素は具体化対応 $\gamma = \alpha^{-1}$ によって具体領域の部分集合に対応付けられる．

上記の定義から，代入・取得・照合ノードの形式を持つ *Boolean* 型の OCL 表現は，具体値 `true` を持つ場合にのみ抽象値 T を持ち，さらに具体値 `false` を持つ場合にのみ抽象値 F を持つため，抽象値によって条件 1_{AG} と条件 1_{C} の充足を判断することが出来る．

3.2.2 抽象領域における解釈規則

抽象領域上では，OCL 表現の種類毎に定めた規則に従って，ノードの意味を解釈する．一例として表 3.4 に *Boolean* 型の論理演算の解釈規則を示す．表の行と列はノードの取りうる抽象値と演算の種類を示し，交点のセルは抽象構文木の子ノードに割り当てるべき抽象値を示す．本手法では 3.2.3 項で述べる方法に従って，抽象構文木の根ノードから順に各ノードの抽象値を下向きに計算するため，抽象値の割り当て規則は親ノードから子ノードへの一方向である．子ノードが複数存在する場合は，構文木と同一の順序で子ノード毎にセルを設けている．OCL の非決定性により，子ノードの抽象値が一意に定まらない場合は，取り得る値を “/” で区切って記載している．例えば子ノード 1 と子ノード 2 の抽象値が各々 “T/U” と “U/T” である場合，子ノード 1 と子ノード 2 の抽象値の組み合わせは (T,U) または (U,T) である．

表 3.4 の解釈規則は，OCL 記述の不完全性を再現する．例えば `and` 演算では，親ノードの抽象値が F のケースにおいて，子ノード 1 に F を割当てる場合は，子ノード 2 に U を割当てる．子ノード 2 に T 又は F を指定する過度な解釈（overspecification）は行わない．

本手法では，OCL メタモデルで定義されている図 2.5 の全 12 種類の OCL 表現のうち，OCL の基本サブセットである BasicOCL パッケージに対応する 9 種類の OCL 表現を扱う．抽象構文木の OCL 表現ノードには，OCL 表現の種類とノード名の組み合わせによって，89 のバリエーションが存在する．本手法では，その中の 57 のバリエーションについて解釈規則を定義し，全体の約 64% を網羅する[*8]．付録 A に 57 のバリエーションと解釈規則の定義方法を記載する．

```
CALCULATE-ABSTRACT-VALUES(T:AST)
 1 solution_set := empty_set
 2 foreach e in Sc
 3   b_set := T.allblocks(e)
 4   b_set.remove(e.inconsistentblocks())
 5   solution_set.add(OBTAIN-POSSIBLE-VALUES(b_set))
 6 return solution_set

OBTAIN-POSSIBLE-VALUES(S:SET(BLOCK))
S1: Disable belows from terminal nodes upward:
      blocks having disabled child blocks, and
      blocks not included in S.
S2: Perform all possible interpretations downward.
S3: Return block-set of all enabled ones.
```

図 3.4. 抽象値の計算アルゴリズム

3.2.3　静的解析

事後条件を満たす抽象構文木上の各 OCL 表現ノードの解釈の組み合わせを求めるアルゴリズム CALCULATE-ABSTRACT-VALUES を図 3.4 に示す．OCL の抽象構文木は DAG の構造を持ち，複数の親ノードからリンクされる合流点ノードは，事後条件中の複数の箇所で参照される変数を表す．あるオブジェクト状態において，各 OCL 表現ノードは同時に複数の値を取り得ないため，合流点ノードに至る複数の経路上で，合流点ノードに同一の値を割り当てる必要がある[*9]．そこで合流点ノードの値の一意性を保証するために，アルゴリズムは DAG の閉路の分岐点と合流点のノードの抽象値と解釈の一意な組み合わせの集合 S_c を作成し（2 行目），S_c の各要素 e について，サブルーチン OBTAIN-POSSIBLE-VALUES を用いて抽象値の可能な割り当てを計算する（5 行目）．その際に，分岐点と合流点のノードの，e に含まれない抽象値と解釈を計算の対象から除外する（3，4 行目）．最後に，各ノードの抽象値と解釈の可能な組み合わせの集合を返す（6 行目）．この実行結果から，条件 1 の充足を確認することができる．

図 2.8 の事後条件の，S_c の或る要素 e_1 についての計算結果を図 3.5 に示す．各ノード[*10]には上下 2 層のブロック群を設けてあり，下層の各ブロック中の文字（T または F）はノードの抽象値を，ブロック間のリンクは子ノードの抽象値の割り当てを表す．上層のブロックは，同一の値を持つ下層のブロック群を代表して，親ノードからのリンクを受ける．上層のブロック中の文字が U の場合は，そのノードの抽象値が U であり，子ノードにも U が割り当てられる．白黒が反転表示されたブロックは，事後条件を満たす解釈の組み合わせに含まれない，無

[*8] 但し，21 のバリエーションの解釈規則は，適用対象の型を *Set* または *Bag* に限定する．

[*9] ただし，一経路上での合流点ノードへの U の割り当ては，その経路上で合流点ノードの値が不定であることを意味し，他の経路上で同ノードに T または F を割当てることが可能である．

[*10] ノードを識別するため，具象構文の字句名と番号を付与してある．具象構文で無名の字句は括弧内に抽象構文の名前を示す．

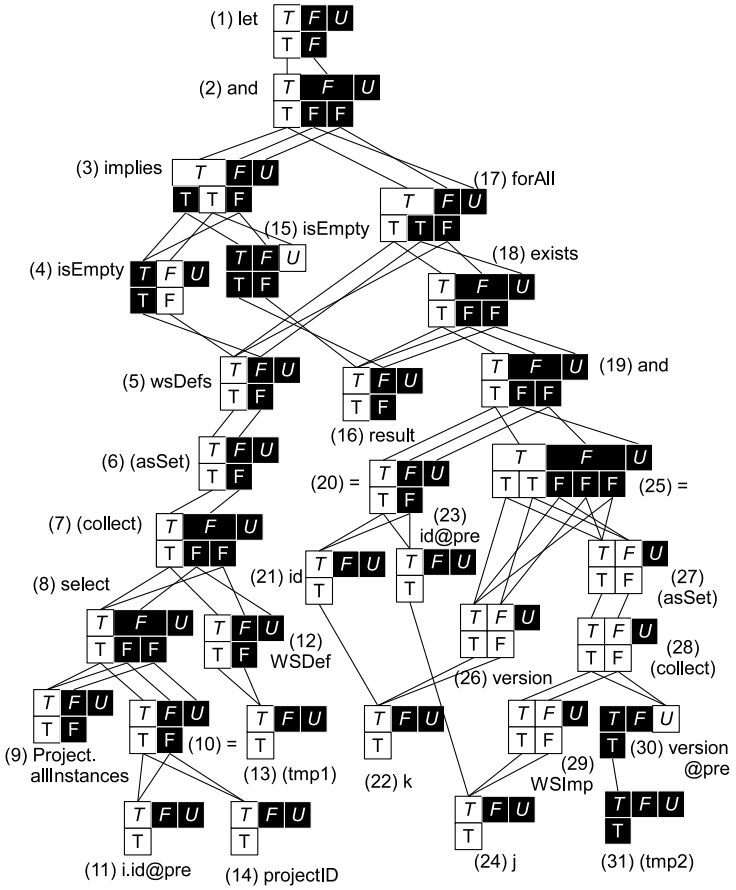


図 3.5. 抽象値の計算結果

効な割り当てであることを示す.

図 3.4 のサブルーチン `OBTAIN-POSSIBLE-VALUES` を説明する．まず `S1` において，末端ノードから順に抽象構文木を上向きに走査し，ノードの各下層ブロックについて，ブロック集合 `S` に含まれない，若しくはリンク先の子ノードのブロックが無効であるようなブロックを無効化する．次に `S2` において，値 T を持つ根ノードの下層ブロックから順に下向きに抽象値の解釈を行い[*11]，可能な解釈の経路を持たないブロックを無効化する．ノードの走査順序を保証するために，`S1`，`S2` では其々，各ノードで全ての子ノード，親ノードからの走査を待ち合わせてから処理を行う．最後に `S3` において，全ての有効なブロックの集合を返す.

本論文では以降，ノード (i) の左端から 0 起点で j 番目の，値 v を持つ下層ブロックを $(i)jv$ のように記述する．図 3.5 は，分岐点ノード (2)，(19) と合流点ノード (5)，(16)，(22)，(24) のブロックの組み合わせ $e_1 = \{(2)0\mathrm{T}, (19)0\mathrm{T}, (5)0\mathrm{T}, (16)0\mathrm{T}, (22)0\mathrm{T}, (24)0\mathrm{T}\}$ についての計算結果を示している．この計算結果から，e_1 において，代入・取得・照合ノードの形式を持つノード (10)，(20)，(25) が何れも抽象値 T，すなわち具体領域で具体値 `true` を取り，条件 1_{AG} と条件 1_{C} を満たすことが分かる．従って，代入ノードの候補集合を $\mathcal{N}_A$，取得ノードの候補

[*11] 事後条件の定義から根ノードは具象値 `true`（抽象値 T）を持つ.

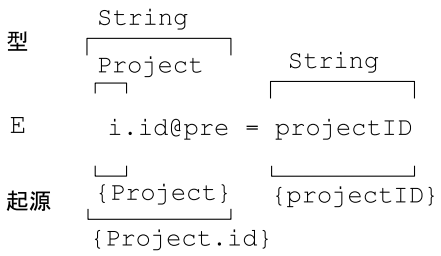


図 3.6. OCL 表現ノードの型と起源

集合を $\mathcal{N}_G$，照合ノードの候補集合を $\mathcal{N}_C$ とすると，$\mathcal{N}_A = \mathcal{N}_G = \mathcal{N}_C = \{(10), (20), (25)\}$ である．

3.3　起源の解析

代入・取得・照合ノードの形式を持つ OCL 表現ノード `E` について，条件 2 を判定するためには，`E` の子ノードの具体的な値が由来するモデル要素を特定する必要がある．子ノードが *Class*/*DataType* 型であれば，由来するモデル要素を型情報から容易に特定することが出来るが，`E` が代入ノード，取得ノード，照合ノードを表す場合は，子ノードは *PrimitiveType* 型もしくはその *Collection* であり，子ノードの型情報からはモデル要素を特定することが出来ない．そこで以下の OCL 表現ノードの起源の概念を導入する．

OCL 表現ノード `E` の起源： `E` の値が由来する構造モデル上の名前付き要素の集合

図 3.6 に，代入・取得・照合ノードの形式を持つノード `E` の具象構文と，その子ノードの型と起源を示す．この例におけるノード `E` は，図 3.5 のノード (10) である．`E` の左右の子ノードの型は *String* であり，代入，取得，照合と無関係な `'a'`，`'b'` のようなリテラルと型の上では区別がつかない．一方，左右の子ノードの起源は其々，`Project` クラスの属性 `id` と，操作パラメータ `projectID` を要素に持つ．これらの起源の情報から，`E` について条件 2 の充足を判定することが出来る．

OCL 表現ノードの起源を求めるアルゴリズム `OBTAIN-ORIGINS` を図 3.7 に示す．`OBTAIN-ORIGINS` は OCL 抽象構文木を再帰的に辿り，引数ノード `exp` の起源である名前付き要素の集合を返す．各 OCL 表現ノードについて求めた起源はメモ化し（9 行目），再利用する（1 行目）．4–8 行目の処理はノード `exp` のタイプに依存する．基本型の定数値を表す *LiteralExp* のインスタンスの場合，構造モデル上に起源を持たないため，5 行目で起源として空リストが設定される．図 3.6 中のノード “`i.id@pre`” のような，属性の参照を表す *PropertyCallExp* ノードの起源は，自身の左の子ノードの起源の要素の，右の子ノードが表す属性から成る．OCL 演算子の呼び出しを表す *OperationCallExp* の中には，子ノード群から起源を継承するものがある（図 3.8）．*Collection* 要素の選択を表す演算 `select` の起源は，子

```
OBTAIN-ORIGINS(exp:OclExpression)
 1  orglist := lookup(exp)
 2  if orglist != null then return orglist
 3  orglist := empty_list
 4  if origins can be obtained from exp itself
 5  then orglist := obtained origins
 6  else explist := child node
 7       foreach i in explist
 8         orglist.concat(OBTAIN-ORIGINS(i))
 9  memorize(exp, orglist)
10  return orglist
```

図 3.7. 起源の解析アルゴリズム

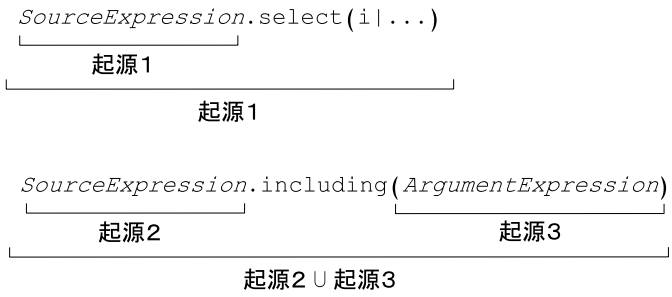


図 3.8. 起源の継承

ノード *SourceExpression* の起源と等しい．また *Collection* へのオブジェクトの追加を表す演算 `including` の起源は，子ノード *SourceExpression*，*ArgumentExpression* の起源の和集合である．他の種類のノードについても同様に，起源の導出方法が定義される．

図 3.5 の代入・取得・照合ノードの形式を持つノード (20)，(25) の左の子ノードの起源は其々，操作の戻り値 `return` の要素の属性 `id`，`version` を要素に持ち，(20)，(25) の右の子ノードの起源は其々，`WSDef` クラスの属性 `id`，`WSImp` クラスの属性 `version` を要素に持つ．従って，図 3.5 のノード (10)，(20)，(25) は，何れも条件 2 を充足する．すなわち，代入ノードの候補集合 $\mathcal{N}_A$，取得ノードの候補集合 $\mathcal{N}_G$，照合ノードの候補集合 $\mathcal{N}_C$ は，$\mathcal{N}_A = \mathcal{N}_G = \mathcal{N}_C = \{(10), (20), (25)\}$ となる．

3.4 データ流向の特定

Step 1.2 で求めた代入ノード，取得ノード，取得ノードの候補のデータ流向を特定することによって，条件 3_{A}，条件 3_{G}，条件 3_{C} の判定を行う方法を説明する．

Step 1.2 の起源の解析で求めた，OCL 表現ノード `E` のオペランドの起源に含まれる操作パラメータとクラスの属性を，それぞれ `param`，`att` とする．本手法では，UML モデルに記

表 3.5. 代入，取得，照合の決定条件

操作パラメータ	*Class* の属性	データ流向	代入	取得	照合
in	`@pre` 有	–	×	×	○
	`@pre` 無	→	○	×	×
out/return	`@pre` 有	←	×	○	×
	`@pre` 無	←	×	○	×

載される `param` の方向（`in/out/return`）と，OCL の操作実行前の値を意味する`@pre` キーワードが `att` に付加されるか否かによって，ノード `E` のデータ流向を特定する．データ流向を特定する上でのポイントは以下の 3 点である．

- `att` に`@pre` が付加される場合には，操作を通じて `att` は変化しない．すなわち，`att` へのデータの流れは存在しない．
- `param` の方向が `in` の場合には，操作を通じて `param` は変化しない．すなわち，`param` へのデータの流れは存在しない．
- `param` の方向が `out/return` の場合には，操作を通じて `param` に値が設定される．すなわち，`param` へのデータの流れが存在する．

上記のポイントから，`param` の方向と，`att` の`@pre` キーワードの有無の組合せによって，Step 1.2 の起源の解析で求めた，OCL 表現ノード `E` のデータ流向を特定して，代入ノード，取得ノード，取得ノードを決定することができる．表 3.5 に，代入ノード，取得ノード，照合ノード照合ノードの決定条件を示す．表中の右矢印（→）は操作パラメータから *Class* の属性へのデータの流れを，左矢印（←）は *Class* の属性から操作パラメータへのデータの流れを，それぞれ表す．

本論文の例題の場合には，Step 1.2 で求めた候補ノード (10)，(20)，(25) の中で，ノード (10) が照合ノードとして，ノード (20)，(25) が取得ノードとして，それぞれ決定される．手法は，Step 1.2 で算出したノード (20)，(25) のオペランドの起源の情報から，これらのノードに関する取得値情報と取得先情報を求めて，代入情報 $A = \phi$，取得情報 $G = \{(\texttt{getAccessibleWS}, \texttt{WSDef}, \texttt{id}, \texttt{result}, \texttt{id}), (\texttt{getAccessibleWS}, \texttt{WSImp}, \texttt{version}, \texttt{result}, \texttt{version})\}$ を出力する．

3.5　確定性の判定

本節では，抽象構文木中の，出力パラメータ又はその属性を表す OCL 表現ノード集合 S_{out} から，常に確定値 T または F を取るノードを，照合出力ノードの候補としてスクリーニングする方法を述べる．

まず S_{out} の構成方法を説明する．操作パラメータとその属性の参照は，各々 *VariableExp* と

```
DECIDE-CERTAINTY(E:NODE,SS:SET(SET(BLOCK)))
 1 foreach S in SS
 2   b_set := S
 3   b_set.removeAll(E.lowerblocks)
 4   solution := OBTAIN-POSSIBLE-VALUES(b_set)
 5   if solution.notEmpty() then return false
 6 return true
```

図 3.9. ノード値の確定性の判定アルゴリズム

PropertyCallExp によって表されるため，これらの種類の OCL 表現ノードを抽出し，さらにその中で出力パラメータまたはその属性を起源とするノードを選んで S_{out} を構成する．各ノードの起源は，Step 1.2 の解析結果を利用する．図 3.5 の例では $S_{out} = \{(16), (21), (22), (26)\}$ である．

次に S_{out} の要素 E の値の確定性を判定する方法を説明する．3.2.2 節で述べた抽象解釈の内容から，E の下層ブロック（値が T 又は F）を含まない有効な割当が存在する場合，かつその場合に限り，ノード E は不定値 U を取る．そこで，ノード E が不定値 U を取り得るか否かを調べることにより，確定性を判定する．

確定性判定アルゴリズム DECIDE-CERTAINTY を図 3.9 に示す．アルゴリズムは，ノード E と，Step 1.1 で計算した S_c の各要素の有効ブロックの組み合わせ集合 SS をパラメータとして動作する．集合 SS の要素である組み合わせ S について，E の下層ブロックを除いたブロック集合 b_set を求め (2, 3 行目)，Step 1.1 の抽象解釈のサブルーチン OBTAIN-POSSIBLE-VALUES を用いて，有効な解釈の組み合わせを求める（4 行目）．そのような組み合わせが存在する場合，E が不定値 U を取ることを意味するので，E の値が確定的ではないと判定する（5 行目）．

S_{out} の要素であるノード (16) について，組み合わせ S' ∈ SS に関する確定性の判定を行った結果を図 3.10 に示す．S' は，Step 1.1 で集合 S_c の要素 $e_2 = \{(2)0\mathrm{T}, (19)2\mathrm{F}, (5)1\mathrm{F}, (16)1\mathrm{F}, (22)0\mathrm{T}, (24)0\mathrm{T}\}$ について計算した，有効な解釈の組み合わせであり，斜線のハッチングで示している．アルゴリズム DECIDE-CERTAINTY の 3 行目で，S' からノード (16) の下層ブロックである (16)1F を除いて計算した結果，S' のブロックは無効となる．従って，白黒反転で示したブロックを含めて，全てのブロックが無効となり，S' に関してノード (16) は不定値 U を取らない．SS の他の要素についても同様に，ノード (16) が不定値 U を取る割り当てが存在しないため，ノード (16) の値が確定的であると判定し，照合出力ノード候補として抽出する．一方，ノード (16) 以外の S_{out} の要素 (21), (22), (26) は，図 3.10 から分かるように，S' において不定値 U を取るため，照合出力ノード候補から除外する．

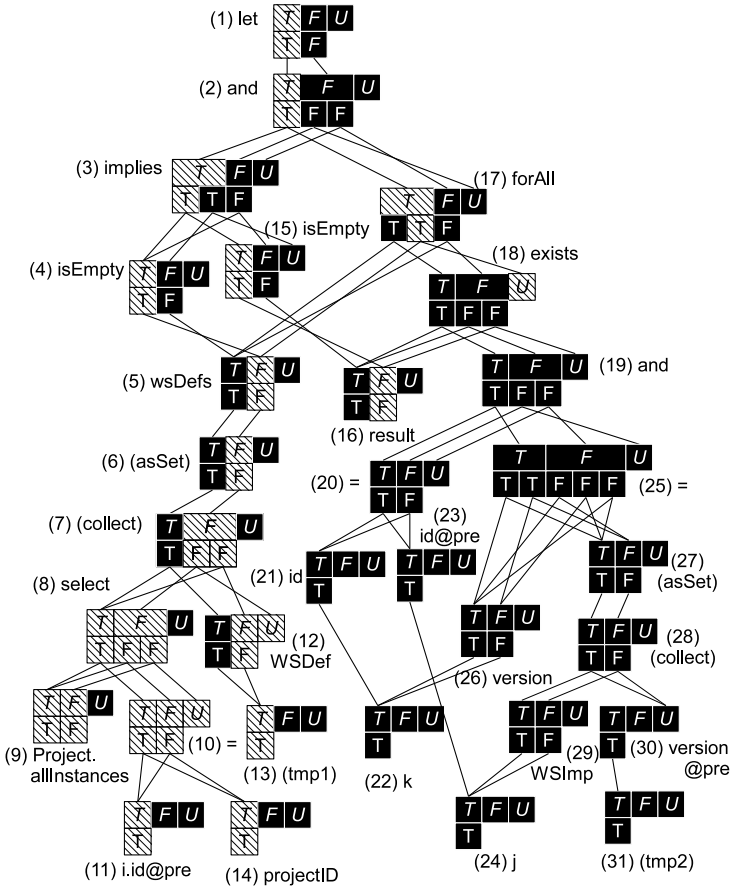


図 3.10. ノード値の確定性の判定の例

3.6　値の対応付け

本節では，Step 2.1 で求めた照合出力ノード候補 $\mathtt{E_0}$ と，Step 1 で求めた照合ノード $\mathtt{E_C}$ の組み合わせについて，$\mathtt{E_0}$ と $\mathtt{E_C}$ の抽象値の間の対応 $f_{\mathtt{E_0},\mathtt{E_C}} : \{\mathrm{T},\mathrm{F}\} \to \underline{D}$ を求める解析方法を述べる．なお照合出力ノード候補 $\mathtt{E_0}$ は常に確定値を取るため，$\mathrm{U} \notin dom(f_{\mathtt{E_0},\mathtt{E_C}})$ である．

本解析方法の工夫は，抽象構文木中のノードの解釈の個々の有効な組み合わせから $f_{\mathtt{E_0},\mathtt{E_C}}$ を求める代わりに，$\mathtt{E_0}$ と $\mathtt{E_C}$ の或る抽象値の組を与える解釈の組み合わせの存在の有無を解析して，$f_{\mathtt{E_0},\mathtt{E_C}}$ を求める点にある．この工夫により，5.1 節で議論するように，実用的なオーダーの計算量で解析を行うことが可能になる．

$f_{\mathtt{E_0},\mathtt{E_C}}$ の計算アルゴリズム `OBTAIN_VALUE_MAPPING` を図 3.11 に示す．アルゴリズムは，ノード $\mathtt{E_0}$，$\mathtt{E_C}$ と，Step 1.1 で求めた S_c の各要素の有効ブロックの組み合わせ集合 `SS` をパラメータとして動作する．まずノード $\mathtt{E_0}$，$\mathtt{E_C}$ の，値 T，F を持つブロック集合 `oT`，`cT`，`oF`，`cF` を求める（3–7 行目）．そして集合 `SS` の各要素 `S` について，Step 1.1 の抽象解釈のサブルーチン `OBTAIN-POSSIBLE-VALUES` を用いて，次の 4 つの有効な解釈の部分集合を求める（9–12

```
OBTAIN-VALUE-MAPPING(Eo,Ec:NODE,SS:SET(SET(BLOCK)))
 1 v_map := empty_map
 2 v_map.put(T,{}), v_map.put(F,{})
 3 oT, oF, cT, cF := empty_set
 4 foreach b in Eo.lowerblocks
 5   if b.value = T then oT.add(b) else oF.add(b)
 6 foreach b in Ec.lowerblocks
 7   if b.value = T then cT.add(b) else cF.add(b)
 8 foreach S in SS
 9   sol_oT := OBTAIN-POSSIBLE-VALUES(S-oF)
10   sol_oT_cU := OBTAIN-POSSIBLE-VALUES(S-oF-cT-cF)
11   sol_oF := OBTAIN-POSSIBLE-VALUES(S-oT)
12   sol_oF_cU := OBTAIN-POSSIBLE-VALUES(S-oT-cT-cF)
13   if sol_oT.intersection(cT).notEmpty()
14   then v_map.get(T).add(T)
15   if sol_oT.intersection(cF).notEmpty()
16   then v_map.get(T).add(F)
17   if sol_oT_cU.notEmpty() then v_map.get(T).add(U)
18   if sol_oF.intersection(cT).notEmpty()
19   then v_map.get(F).add(T)
20   if sol_oF.intersection(cF).notEmpty()
21   then v_map.get(F).add(F)
22   if sol_oF_cU.notEmpty() then v_map.get(F).add(U)
23 return v_map
```

図 3.11. 値の対応付けアルゴリズム

行目)．

- sol_oT： E_0 の値が T の組み合わせ
- sol_oT_cU： E_0 と E_C の値が其々 T，U の組み合わせ
- sol_oF： E_0 の値が F の組み合わせ
- sol_oF_cU： E_0 と E_C の値が其々 F，U の組み合わせ

sol_oT $\cap$ cT $\neq \phi$ の場合，E_0 と E_C の値が共に T であるケースが存在することを意味するので，f_{E_0,E_c} を表す v_map に対 (T,T) を追加する（13，14 行目）．同様に sol_oT $\cap$ cF $\neq \phi$ の場合は対 (T,F) を追加し（15，16 行目），sol_oF $\cap$ cT $\neq \phi$ の場合は対 (F,T) を追加し（18，19 行目），sol_oF $\cap$ cF $\neq \phi$ の場合は対 (F,F) を追加する（20，21 行目）．また sol_oT_cU $\neq \phi$ の場合には，E_0 と E_C の値が其々 T，U であるケースが存在することを意味するので，v_map に対 (T,U) を追加する（17 行目）．同様に sol_oF_cU $\neq \phi$ の場合は対 (F,U) を追加する（22 行目）．最後に v_map を f_{E_0,E_c} として返す（23 行目）．

照合出力ノード候補 (16) と照合ノード (10) について，有効な解釈の組み合わせ S' $\in$ SS に関するブロック集合 sol_oF_cU の計算例を図 3.12 に示す．図中の白抜きの有効なブロック集

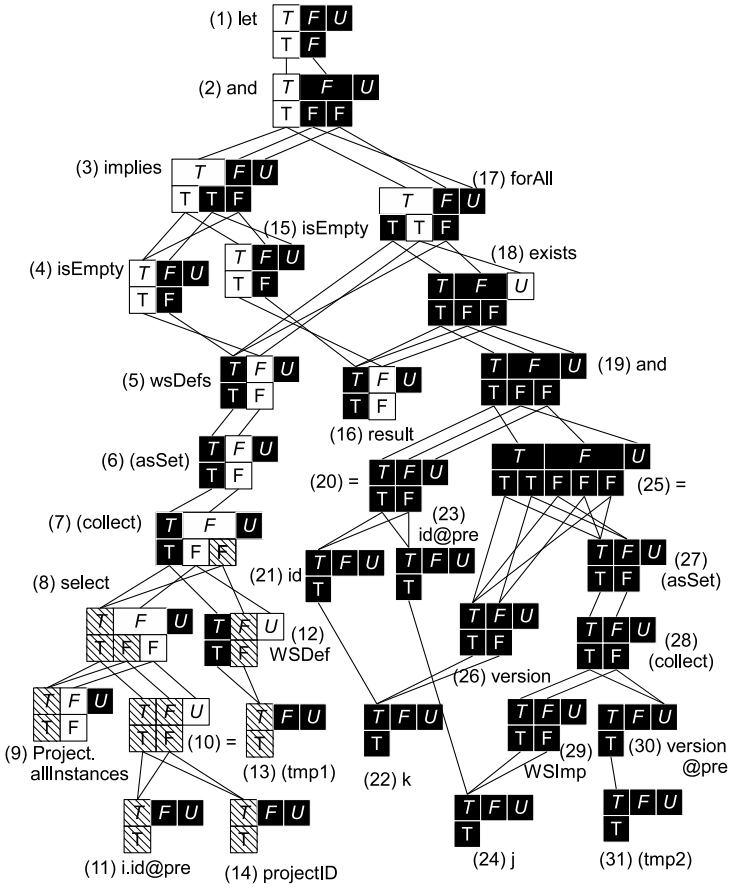


図 3.12. 値の対応付けの例

合が sol_oF_cU であり，斜線のハッチングは，アルゴリズム OBTAIN-VALUE-MAPPING の 12 行目で，S’ からノード (10) の下層ブロックである (10)0T と (10)1F を除いて計算した結果，無効となったブロック集合 S’-sol_oF_cU を示している．図 3.12 の計算結果から，sol_oF_cU $\neq \phi$ であり，$\mathtt{E_0}$ と $\mathtt{E_C}$ の値が其々 F, U である組み合わせが存在するため，$f_{(16),(10)}(\mathrm{F}) \ni \mathrm{U}$ となる．本アルゴリズムを実行することにより，最終的に $f_{(16),(10)}(\mathrm{T}) = \{\mathrm{T}\}$, $f_{(16),(10)}(\mathrm{F}) = \{\mathrm{T,F,U}\}$ が得られる．

3.7　含意判定

本節では，Step 2.2 で求めた対応 $f_{\mathtt{E_0},\mathtt{E_C}}$ から，照合出力ノード候補 $\mathtt{E_0}$ の値と，照合結果が真であることの間の含意判定を行う方法を述べる．

3.1.3 項で述べた，照合出力ノードの満たすべき条件 6 から，照合出力ノードの或る出力値に対して，照合結果が一意に T に定まる．本手法では，そのような出力値の抽象値が，T，F の何れか一方に定まるという制限を設け，次の条件式の成立を判定する（$\mathtt{V_0}$ を照合出力ノード

表 3.6. 含意の成立パターン

出力値 $\mathtt{v_0} \in \mathtt{V_0}$	$f_{\mathtt{E_0},\mathtt{E_c}}(\mathtt{v_0})$	
T	{T}	S$\in \mathfrak{P}$({T,F,U})\{T}
F	S$\in \mathfrak{P}$({T,F,U})\{T}	{T}

候補 $\mathtt{E_0}$ の抽象値の集合とする).

$$\exists v \in \mathtt{V_0}(\forall v' \in \mathtt{V_0}(v = v' \Leftrightarrow f_{\mathtt{E_0},\mathtt{E_c}}(v') = \{\mathrm{T}\}))$$

上記の式を満たす 2 つのパターンを表 3.6 に示す．第 1 のパターンでは $f_{\mathtt{E_0},\mathtt{E_c}}(\mathrm{T}) = \{\mathrm{T}\}$ であり，第 2 のパターンでは $f_{\mathtt{E_0},\mathtt{E_c}}(\mathrm{F}) = \{\mathrm{T}\}$ である．記号 $\mathfrak{P}$ は冪集合を表す．本手法では，$f_{\mathtt{E_0},\mathtt{E_c}}$ が表 3.6 のパターンの何れかに合致する場合に，$\mathtt{E_0}$ を照合出力ノードと判定する．

例題の，照合出力ノード候補 (16) と照合ノード (10) の組合せの場合には，Step 2.2 で求めた $f_{(16),(10)}$ が第 1 のパターンに該当する．手法は，Step 1.2 で算出したノード (10)，(16) のオペランドの起源の情報から，照合先情報，照合値情報，照合出力情報を求めて，照合情報 $C = \{(\mathtt{getAccessibleWS}, \mathtt{Project}, \mathtt{id}, \mathtt{projectID}, \mathtt{NULL}, \mathtt{result}, \mathtt{NULL})\}$ を出力する．

3.8 支援ツール

OCL 記述の解析手法の適用を支援するために作成したツールの説明を行う．ツールは，解析部と表示部の 2 つの部分から構成される（図 3.13）．解析部は，コンポーネントの仕様モデルが記述された XMI 形式のファイル[*12] を入力として，3.2–3.7 節で説明した Step 1 と Step 2 の解析を自動で実行し，DC の操作の事後条件ごとに，解析結果，抽象構文木の解釈規則，抽象値の計算結果を出力する．一方，表示部は，解釈規則と抽象値の計算結果を可視化する．

解析結果には，各サブステップ毎の計算結果がテキスト形式で記載される．記号 $*$ が付加された部分は，該サブステップの条件を充足することを表す情報である．ツールの利用者は，解析結果の内容をサブステップ毎に順を追って参照することによって，解析の途中経過と最終結果を知ることが出来る．

解釈規則は，3.2.2 項で説明した，事後条件の親子ノードの抽象値の割り当て規則の情報であり，抽象値の計算結果は，3.2.3 項で説明した，抽象値の可能な割り当て情報である．後者の割り当て情報は，構文木の分岐点ノードと合流点ノードの解釈の組み合わせ e の各々について，各ノードが取り得る抽象値の組み合わせが，テキストファイルの一行に記載される．ツールの利用者は，grep 等のプログラムを用いて，抽象値の計算結果のパターンマッチングを行い，関心のあるノード値の組み合わせを抽出することが出来る．例えば “(16)[0-9]T” と “(10)[0-9]T” を含む行を抽出し，照合が成功する場合のノード値の組み合わせを得ることが出来る．ただし [0-9] は数字 1 文字を表すものとする．

[*12] XMI は，UML モデルの交換に用いられる標準的なファイル形式である．

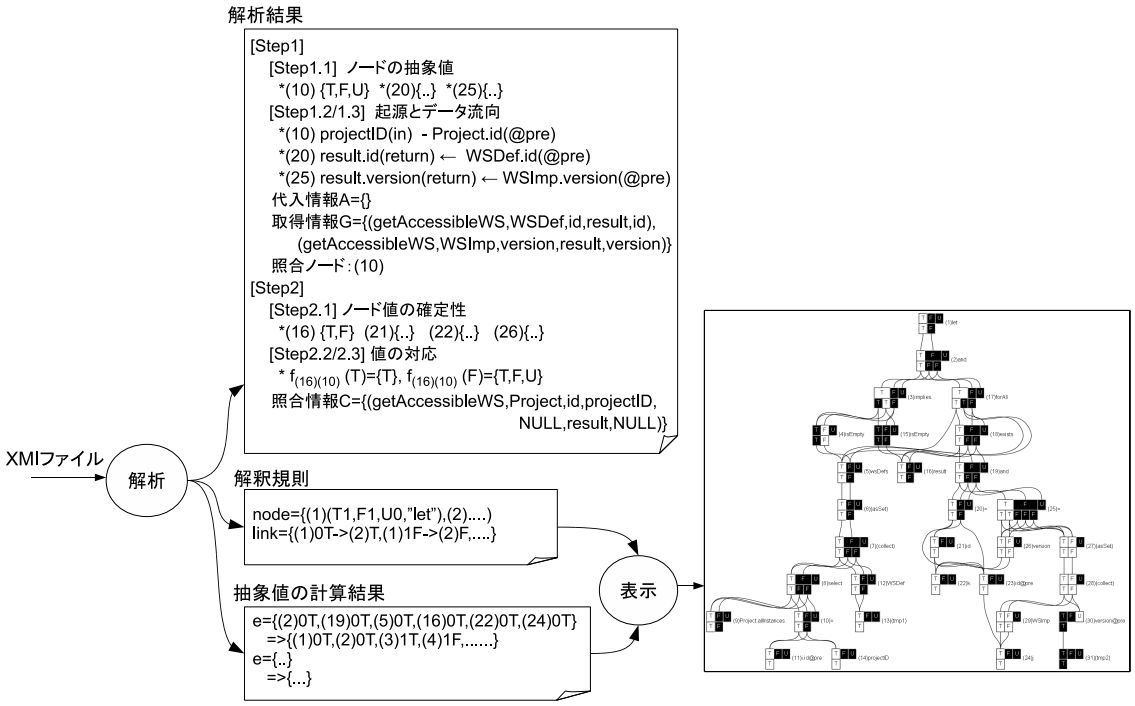


図 3.13. 支援ツール（OCL 記述の解析）

或る e のノード値の組み合わせと解釈規則を入力として，ツールの表示部を実行すると，e における有効な解釈の組み合わせが図 3.5 の形式のグラフとして表示される．解釈規則のみを入力すると，Step 1.3 の静的解析を実行する前の，全てのノードが有効なグラフが表示される．これらのグラフ表示は，解析結果をより深く理解するために役立つ．例えば，利用者が手動で抽出した代入情報，取得情報，照合情報の内容が，ツールの解析結果と異なる場合に，グラフを詳細に調べることにより，利用者とツールの間で解釈の違いが生じた OCL 表現ノードを探すことが出来る．

本研究では，Java 言語の UML 処理系である MDT/UML2（MDT project [126]）と OCL 処理系の MDT/OCL（MDT project [127]）を利用して，ツールの解析部の実装を行った．また表示部の実装では，グラフ描画プログラムである Graphviz（AT&T Research [128]）を利用した．

第 4 章

Activity の解析

本章では，PC の振舞いにおける DC の操作の利用手順の整合性（整合性 2）の判定を支援するための Activity の解析手法の説明を行う．2.4.4 項で述べたように，PC の振舞いにおける DC の操作の利用手順の整合性 2 の判定では，合成モデルの $Activity$ 中のデータアクセスの手順とデータ伝搬に関する複雑な条件判定が必要であり，人手で網羅的かつ正確に判定を行うことが難しい．本解析手法は，PC の振舞いモデルを静的に解析して，整合性 2 の判定を自動で実行する．本解析手法の出力情報を利用することにより，整合性判定の正確性と効率の向上が可能になる．

4.1 節で解析の概要を述べる．4.2–4.5 節において，解析の 4 つの手順（コールグラフの生成，破壊的アクセス・補完的アクセスの定義，データフローの解析，整合性判定）を説明する．4.6 節で，手法の適用を支援するツールの説明を行う．

4.1 概要

DC 間に跨る参照の整合性を保証するために，参照先と参照元の DC のオブジェクト状態の同期が必要である．整合性 2 とは，オブジェクト状態の同期のために，PC の振舞いの中で参照先と参照元の DC のデータアクセス操作が適切な手順で利用されることである．

システムの各機能は複数の PC，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，整合性 2 が保たれなければならない．Activity の解析手法は，システムが提供する機能ごとに，振舞いの合成モデルの中で起こりうる DC のデータアクセス操作の利用手順の正しさを判定する．複数機能を同時実行する場合の並行性に関する判定は行わない．

4.1.1 解析の入出力

本解析は，DC と PC の構造モデル，PC の振舞いモデル，アクセス情報 AC の構成要素である代入情報 A，取得情報 G，削除情報 D，照合情報 C を入力として，参照の集合 REF に対する PC の振舞いモデルの整合性判定を行い，その結果を出力する（図 4.1）．

アクセス情報 $AC = (A, G, D, C)$ は，3.1.1 項で導入した，DC の振舞いモデルのデータア

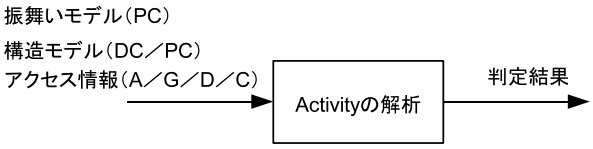


図 4.1. Activity の解析の入出力

クセスに関する性質を示す情報である．UML ではアクセス情報 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 の構成要素である代入情報 A，取得情報 G，照合情報 C，削除情報 D の内容をテキスト形式で記述する．また，システム機能を提供する PC の操作は，命名規則やステレオタイプの付与などの手段を用いて特定する．

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．従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，判定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，十分な判定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全ての PC，DC の仕様を入力として与える必要がある．

4.1.2　技術課題と解決アプローチ

データアクセス手順の判定を行う上での技術課題と，本解析手法で採用する課題解決アプローチを表 4.1 にまとめる．

2.4.4 項で述べたように，参照整合性を実現するデータアクセス手順は一様ではない．合成モデル中のデータアクセス順序の正しさを判断するための基準が必要である（技術課題 5）．2.5.2 項で説明した Planas et al. [107, 108] は，データ整合性を破壊する可能性がある *Action* の種別毎に，整合性を満たすために必要な補完 *Action* を定義した．本手法では，Planas らのアイデアを複数の DC 間に拡張して，DC 間の参照整合性を破壊しうるアクセス（破壊的アクセス）と，破壊を防ぐために必要なアクセス（補完的アクセス）の組を定義し，破壊的アクセスと補完的アクセスの順序を定めることによって，技術課題 5 を解決する．このアプローチは，2.5.5 項で説明した関係データベースの参照整合性の実現方法の類比である．

2.4.4 項で述べたとおり，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，異なる DC のデータ間の対応関係を確認するためには，図 2.11 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 6）．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，

表 4.1. Activity 解析：課題解決のアプローチ

	技術課題	解決のアプローチ
5	データアクセス順序の判断基準	破壊的アクセスと補完的アクセスの組の定義
6	アクセス操作の引数の由来の特定	Field-sensitive なデータフロー解析
7	制御とデータの流れに依存する条件の判定	判定条件の緩和（データフローに基づく判定）

図 4.2. Activity の解析手順

操作呼び出しの引数の到達定義を求めることにより，技術課題 6 を解決する．

整合性 2 を判定するためには，破壊的アクセスと補完的アクセスの操作に与える引数の同値関係を調べる必要がある．しかし，同値関係の成立は *Activity* の制御とデータの流れに依存するため，そのような複雑な条件判定を，静的な解析によって厳密に行うことは困難である（技術課題 7）．本解析では，データフローに基づいて同値関係の成立の可能性を求めて，緩和された判定条件として用いることによって，技術課題 7 を解決する．

4.1.3　解析手順

Activity の解析手順を図 4.2 に示す．本解析手法は，PC，DC の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，整合性の判定を行う Step 4 の 4 ステップからなる（図 4.2）．全システム機能に対して Step 1，2 を 1 度ずつ実施した後，システム機能ごとに Step 3，4 を繰り返し実施する．

Step 2 で，2.5.5 項で述べた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，技術課題 5 を解決する．Step 3 で，Field-sensitive な *Activity* のデータ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，技術課題 6 を解決する．Step 4 では，*Activity* 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，技術課題 7 を解決する．

4.2–4.5 節で，Step 1–4 の各ステップの詳細な内容を説明する．

4.2　コールグラフの生成

各システム機能は，複数の PC，DC の操作の振る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，DC 間にまたがる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，システム機能毎の PC，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 4.3 に示す．$call = (op, op', a') \in CG$ は，PC の操作 op’ の振る舞いを記述する $Activity$ 中の $a' \in CallOperationAction$ が，他の PC 又は DC の操作 op を呼び出すことを表す．

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

図 4.3. コールグラフ CG

システム機能 F の CG を次の手順で静的に生成する．先ず空の CG に，F を提供する PC の操作 op' と，その振る舞いを表す $Activity$ 中で呼び出す操作 op の組を追加する．op が PC 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 2.15，図 2.16 の例題では，操作 `WSExecutionAgent::sysExecuteWS` が提供する唯一のシステム機能について $CG = \{(\texttt{getAccessibleWS}, \texttt{sysExecuteWS}, 3), (\texttt{executeWS}, \texttt{sysExecuteWS}, 15), (\texttt{addAccessor}, \texttt{executeWS}, 25)\}$ が得られる．ただし 3，15，25 は，各番号に対応するノードを表す．

4.3　破壊的アクセス・補完的アクセスの定義

本手法では，AC が表す DC のデータアクセスの中で，DC 間に跨る参照整合性を破壊し得るアクセスと，破壊を防ぐ為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わせを定義する．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．

2.3.2 項で述べたように，本手法では参照先のキー属性 $prop_{dst}$ は cls_{dst} の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と想定する．従って，実行パターン (3) の $prop_{dst}$ の変更は発生しないため，(1) と (2) の実行パターンを扱う．

データアクセスの実行パターンは，PC 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 4.2 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，PC 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 つの何れかのアクセスを実行す

表 4.2. 参照 r の破壊的アクセス・補完的アクセス

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)	$D_r^{src} = \{(op, cls, prop_{cls}, param, prop_{param}) \in D \mid prop_{cls} = r.prop_{src}\}$

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) 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．

上記の実行パターン (1-2) は，2.5.5 項で述べた，関係データベースの参照表の外部キー値の更新を行う際の，参照制約動作の CASCADE あるいは RESTRICT による被参照表の照合チェックの類比である．また実行パターン (2-1)，(2-2) は，被参照表のレコードの削除を行う際の，RESTRICT による参照表の照合チェックと，CASCADE による連鎖削除の類比である．その他の (1-1) と (1-3) は，参照制約動作を用いずに，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，PC 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 の破壊を防ぐ為に実行される，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其々，表 4.2 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 2.15 の例題では，参照 $\mathtt{r_1}$ に関して，$(\mathtt{addAccessor}, \mathtt{Accessor}, \mathtt{projectID}, \mathtt{projectID}, \mathtt{NULL}) \in A$, $(\mathtt{getAccessibleWS}, \mathtt{Project}, \mathtt{id}, \mathtt{projectID}, \mathtt{NULL}, \mathtt{return}, \mathtt{NULL}) \in C$ 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，$A^{src}_{\mathtt{r_1}} = \{(\mathtt{addAccessor}, \mathtt{Accessor}, \mathtt{projectID}, \mathtt{projectID}, \mathtt{NULL})\}$，$C^{dst}_{\mathtt{r_1}} = \{(\mathtt{getAccessibleWS}, \mathtt{Project}, \mathtt{id}, \mathtt{projectID}, \mathtt{NULL}, \mathtt{return}, \mathtt{NULL})\}$，$D^{dst}_{\mathtt{r_1}} = G^{dst}_{r_1} = A^{dst}_{r_1} = C^{src}_{r_1} = D^{src}_{r_1} = \phi$ である．図 2.16 の例題では，参照 $\mathtt{r_2}$ に関して，$(\mathtt{addAccessor}, \mathtt{WSInfo}, \mathtt{wsDefID}, \mathtt{wsDefID}, \mathtt{NULL}) \in A$, $(\mathtt{getAccessibleWS}, \mathtt{WSDef}, \mathtt{id}, \mathtt{return}, \mathtt{id}) \in G$ が，それぞれ (1) の破壊的アクセス，(1-1) の補完的アクセスであり，$A^{src}_{\mathtt{r_2}} = \{(\mathtt{addAccessor}, \mathtt{WSInfo}, \mathtt{wsDefID}, \mathtt{wsDefID}, \mathtt{NULL})\}$，$G^{dst}_{\mathtt{r_2}} = \{(\mathtt{getAccessibleWS}, \mathtt{WSDef}, \mathtt{id}, \mathtt{return}, \mathtt{id})\}$，$D^{dst}_{\mathtt{r_2}} = C^{dst}_{r_2} = A^{dst}_{r_2} = C^{src}_{r_2} = D^{src}_{r_2} = \phi$ である．

4.4　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，4.3 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* [*1]，或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．例えば，図 2.15 の場合には，操作 `getAccessibleWS` の呼び出し（手順 B）の *InputPin*（番号 2）と，操作 `addAccessor` の呼び出し（手順 E）の *InputPin*（番号 22）の値が等しくなければならない．図 2.16 の場合には，操作 `getAccessibleWS` の呼び出し（手順 F）の *OutputPin*（番号 4）の属性 `id` と，操作 `addAccessor` の呼び出し（手順 J）の *InputPin*（番号 24）の値が等しくなければならない．

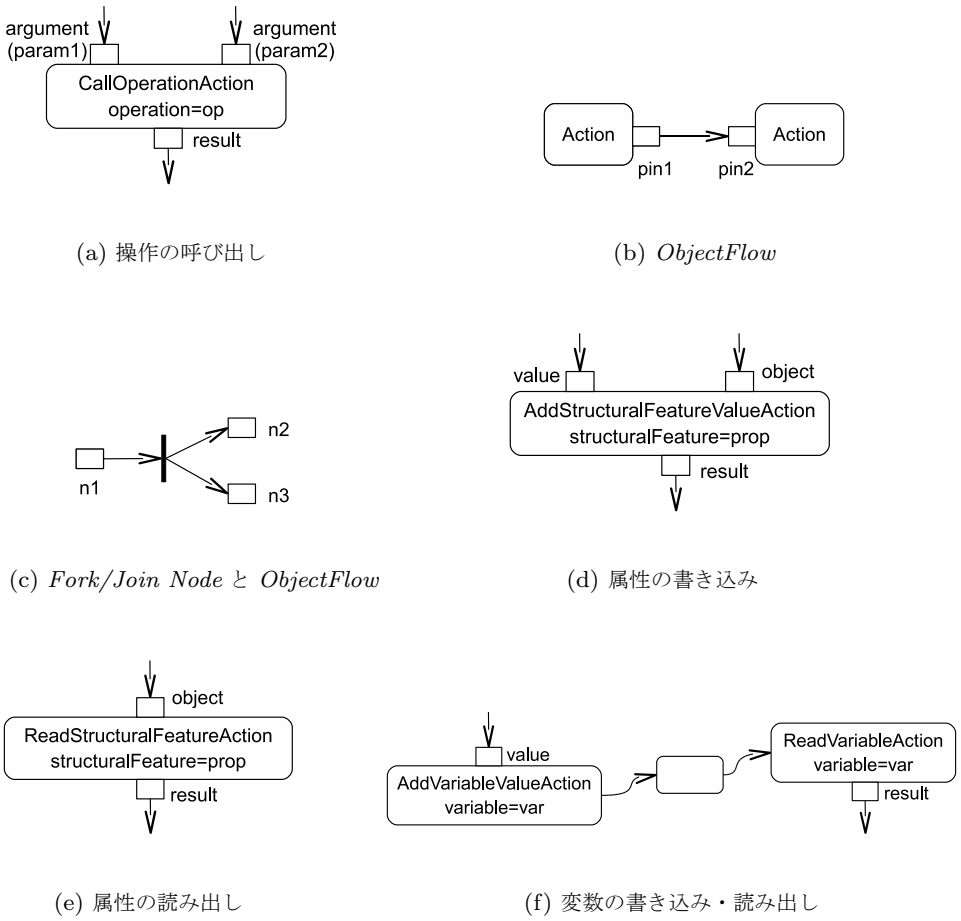
操作呼び出しの引数と戻り値の関係は，*Activity* の *ObjectNode* 或いはその属性の間のデータの伝搬によってモデル化される．そこで本ステップでは，field-sensitive なデータフロー解析を行い，操作呼び出しの引数の到達定義を求める．本ステップで求めた到達定義の情報を用いて，次の Step 4 の中で，操作呼び出しの引数と戻り値の同値関係の判定を行う．

データの伝搬は複数の *Activity* に跨る可能性がある．本ステップでは各 *Activity* 内の *ObjectNode* 間の依存解析を行った後，*Activity* 間に跨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* 毎に一度，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．

4.4.1　ObjectNode 間の依存解析

Activity 内の *ObjectNode* 間の依存関係 $IN \subseteq (ObjectNode \times (Property \cup \mathtt{NULL})) \times$

[*1] 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 4.4(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．



(a) 操作の呼び出し　(b) *ObjectFlow*

(c) *Fork/Join Node* と *ObjectFlow*　(d) 属性の書き込み

(e) 属性の読み出し　(f) 変数の書き込み・読み出し

図 4.4. *Activity* におけるデータの伝搬

$(ObjectNode \times (Property \cup \texttt{NULL}))$ を求める．*IN* の定義を図 4.5 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 $p' = \texttt{NULL}$ 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，*ObjectFlow* によって表現される．図 4.4(b) は，2 つの *ObjectNode* 間の *ObjectFlow* を示している．*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．さらに，図 4.4(c) のように，*Fork/Join Node* と *ObjectFlow* の組合せによって伝搬が行われる場合がある．*IN* の定義では，n' からの *ActivityEdge* と n への *ActivityEdge* を持つような *Fork/Join Node* が存在するという条件（4–5 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 4.4(d) に示す属性の書き込み（*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

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

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

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

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

図 4.5. *ObjectNode* 間の依存関係 *IN*

が，`result` ノードの対応する属性に伝搬する．一方，図 4.4(e) 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 4.5 の *IN* の定義では，6–8 行目の論理式によって前者の書き込みの場合を，9–10 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・読み出しの組合わせによって表現される．図 4.4(f) 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 4.5 の *IN* の定義では，11–13 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，或いは $n,n' \in Action$ であり，かつ n' に接続された *OutputPin* から，n に接続された *InputPin* へ

$$DEF = PG^{+}$$

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

図 4.6. 到達定義 DEF

の $ObjectFlow$ が存在するケースの何れかに該当する．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 4.5 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，或いは n' での書き込みが，n の或る直前ノード n'' を介して n に到達するケースの何れかである．従って $REACH$ 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，$REACH$ を求めることが出来る．

4.4.2 到達定義の解析

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

DEF，PG の定義を図 4.6 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 4.6 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．

表 4.3. データフロー解析の結果

集合	値
$KILL$	ϕ
CF	{(3,1),(5,3),(7,5),(10,7),(15,10),(16,15),(25,21),(26,25)}
$REACH$	ϕ
IN	{((2,NULL),(17,NULL)),((14,NULL),(17,NULL)),((6,id),(19,id)),((6,version),(19,version)), ((8,NULL),(6,id)),((9,id),(19,id)),((9,version),(19,version)),((11,NULL),(9,version)), ((13,NULL),(8,NULL)),((12,NULL),(11,NULL)),((22,NULL),(27,NULL)),((23,NULL),(28,NULL)), ((24,NULL),(29,NULL))}
$DEF_{(2,\texttt{NULL})}$	{(17,NULL)}
$DEF_{(4,\texttt{id})}$	ϕ
$DEF_{(22,\texttt{NULL})}$	{(27,NULL),(14,NULL),(17,NULL)}
$DEF_{(24,\texttt{NULL})}$	{(29,NULL),(13,NULL),(8,NULL),(6,id),(19,id)}

図 2.15，図 2.16 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 4.3 の $KILL$, CF, $REACH$, IN, DEF が得られる．ただし表 4.3 では，DEF については，Step 4 の整合性検証の際に利用する，番号 2, 4, 22, 24 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 $(n,p) \in ObjectNode \times (Property \cup \texttt{NULL})$ に対して定義され，$DEF_{(n,p)} = \{(n',p') \in ObjectNode \times (Property \cup \texttt{NULL}) | ((n,p),(n',p')) \in DEF\}$ である．

4.5　整合性判定

本ステップでは，或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 $Activity$ の整合性の判定を行う．なお以下では，4.3 節で述べた実行パターン (1) と (1-1)，(1-2)，(1-3) の組み合わせについて説明するが，(2) と (2-1)，(2-2) の組み合わせについても同様の手順で判定を行うことが出来る．

4.5.1　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，アクセス種別 $ACTYPE \subseteq REF \times CallOperationAction \times (A \cup G \cup C)$ を求める．$HIST$, $ACTYPE$ の定義を図 4.7 に示す．

$(r,n,n') \in HIST$ は，$Activity$ 中の $n \in ControlNode \cup ExecutableNode$，或いは 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 4.7 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 $Activity$ 間に跨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．

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

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

図 4.7. アクセス履歴 $HIST$ とアクセス種別 $ACTYPE$

$HIST$ 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出来る．

$(r,n,e) \in ACTYPE$ 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．

図 2.15，図 2.16 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 4.4 の $HIST$ と $ACTYPE$ が得られる．

表 4.4 の $ACTYPE$ の値と，4.3 節の説明で述べた $A_{\mathtt{r}_1}^{src}$，$C_{\mathtt{r}_1}^{dst}$ の値から，番号 3，25 のノードで，それぞれ r_1 の補完的アクセス (`getAccessibleWS`, `Project`, `id`, `projectID`, `NULL`, `return`, `NULL`) $\in C_{\mathtt{r}_0}^{dst}$，破壊的アクセス (`addAccessor`, `Accessor`, `projectID`, `projectID`, `NULL`) $\in A_{\mathtt{r}_0}^{src}$ の操作呼び出しが行われることが分かる．

同様に，表 4.4 の $ACTYPE$ と $A_{\mathtt{r}_2}^{src}$，$G_{\mathtt{r}_2}^{dst}$ の値から，番号 3，25 のノードで，それぞれ r_2 の補完的アクセス (`getAccessibleWS`, `WSDef`, `id`, `return`, `id`) $\in G_{\mathtt{r}_2}^{dst}$，破壊的アクセス (`addAccessor`, `WSInfo`, `wsDefID`, `wsDefID`, `NULL`) $\in A_{\mathtt{r}_2}^{src}$，の操作呼び出しが行われることが分かる．

表 4.4. アクセス履歴とアクセス種別の計算結果

集合	値
$ACTYPE$	{(r1,3,(getAccessibleWS, Project, id, projectID, NULL, return, NULL)), (r1,25,(addAccessor, Accessor, projectID, projectID, NULL)), (r2,3,(getAccessibleWS, WSDef, id, return, id)), (r2,25,(addAccessor, WSInfo, wsDefID, wsDefID, NULL))}
$HIST$	{(r1,3,3),(r1,5,3),(r1,7,3),(r1,10,3),(r1,15,3),(r1,16,3),(r1,21,3),(r1,25,3), (r1,26,3),(r1,25,25),(r1,26,25),(r1,15,25),(r1,16,25), (r2,3,3),(r2,5,3),(r2,7,3),(r2,10,3),(r2,15,3),(r2,16,3),(r2,21,3),(r2,25,3), (r2,26,3),(r2,25,25),(r2,26,25),(r2,15,25),(r2,16,25)}

4.5.2　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 4.8 に示す．Step 2 の説明で述べたように，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[*2][*3]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，$h.n'$ による破壊を防ぎ得ることを基準として，整合性を判定する（5–19 行目）．

ここで 4.3 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[*2*3]，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．

```
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 4.8. 整合性の判定アルゴリズム

[*2] 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 4.8 の 4，9 行目でそれぞれ定義されている．

[*3] 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 4.2 の内容に従って計算される．

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

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： $t'.e \in G_r^{dst}$ の時，$o_{h'.n'}$ 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．即ち $o_{h.n'}$ 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： $t'.e \in A_r^{dst}$（$t'.e \in C_r^{dst}$）の時，$o_{h.n'}$ 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．即ち，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 4.8 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} が互いに素の時に，判定失敗と出力する（19 行目）．

表 4.3，表 4.4 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．VERIFY の 1 行目以降で，$\mathtt{r_1}, \mathtt{r_2} \in REF$ のそれぞれについて判定が行われる．

まず $\mathtt{r_1}$ のケースを説明する．$\mathtt{r_1}$ に関して，$HIST$ 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 $(\mathtt{r_1}, 25, 25), (\mathtt{r_1}, 26, 25), (\mathtt{r_1}, 15, 25), (\mathtt{r_1}, 16, 25)$ の 4 つである．これらの何れについても $d_{reach} = DEF_{(22,\mathtt{NULL})}$ 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 $(\mathtt{r_1}, 25, 3)$，$(\mathtt{r_1}, 26, 3)$，$(\mathtt{r_1}, 15, 3)$，$(\mathtt{r_1}, 16, 3)$ から $c_{reach} = DEF_{(2,\mathtt{NULL})}$ となる．$d_{reach} \cap c_{reach} = \{(17, \mathtt{NULL})\} \neq \phi$，すなわち番号 22 のノードと，番号 2 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，アルゴリズムは判定成功を出力する．

次に $\mathtt{r_2}$ のケースを説明する．$\mathtt{r_2}$ に関して，$HIST$ 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 $(\mathtt{r_2}, 25, 25)$，$(\mathtt{r_2}, 26, 25)$，$(\mathtt{r_2}, 15, 25)$，$(\mathtt{r_2}, 16, 25)$ の 4 つである．これらの何れについても $d_{reach} = DEF_{(24,\mathtt{NULL})}$ 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 $(\mathtt{r_2}, 25, 3)$，$(\mathtt{r_2}, 26, 3)$，$(\mathtt{r_2}, 15, 3)$，$(\mathtt{r_2}, 16, 3)$ から $c_{reach} = DEF_{(4,\mathtt{id})}$ となる．$d_{reach} \cap c_{reach} = \phi$，すなわち番号 24 のノードの到達定義に，番号 4 のノードの属性 `id` が含まれないため，判定の条件 1 が成立せず，アルゴリズムは判定失敗を出力する．

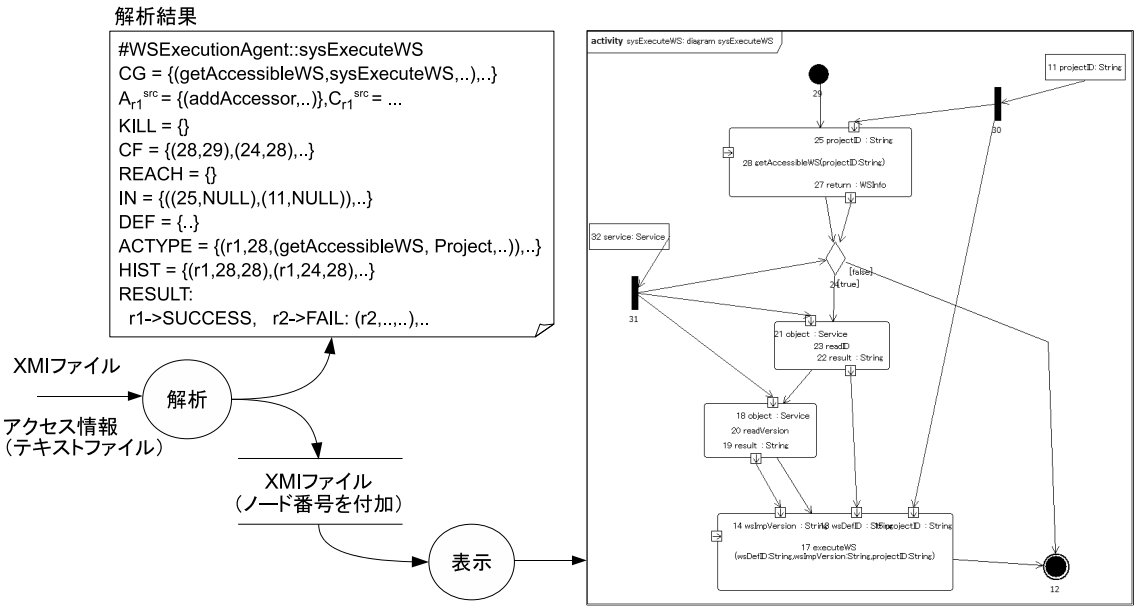


図 4.9. 支援ツール（Activity の解析）

4.6　支援ツール

Activity の解析手法の適用を支援するために作成したツールについて説明を行う．ツールは，解析部と表示部の 2 つの部分から構成される（図 4.9）．解析部は，コンポーネントの仕様モデルが記述された XMI 形式のファイルと，アクセス情報が記載されたテキストファイルを入力として，4.2–4.5 節で説明した Step 1–4 の解析を自動で実行し，システム機能を提供する PC の操作ごとの解析結果と，*Activity* の各ノードに番号を付加した XMI ファイルを出力する．表示部は，ノード番号を付加した *Activity* を可視化する．

解析結果には，中間ステップの計算結果（コールグラフ *CG*，破壊的アクセスと補完的アクセス，変数の無効化関係 *KILL* と到達関係 *REACH*，制御の順序関係 *CF*，依存関係 *IN*，到達定義 *DEF*，アクセス種別 *ACTYPE*，アクセス履歴 *HIST*）と，参照ごとの整合性の判定結果が，テキスト形式で記載される．判定失敗のケースでは，判定の根拠となった破壊的アクセスの操作呼び出しの履歴と，その履歴についての d_{reach} と c_{reach} を，詳細情報として出力する．ツールの利用者は，解析結果の内容をステップ毎に順を追って参照することによって，解析の途中経過と最終結果を把握することが出来る．

解析結果をテキスト形式で提示するために，*Activity* 中のノードを一意に識別する文字が必要であるが，解析対象のモデルには，そのような識別文字が含まれない[*2]．そこで，ツールの

[*2] XMI ファイル中の要素には一意な id が付与されるが，この id は，UML モデルをシリアライズするために XMI のスキーマで規定されているものであり，ユーザが UML モデルの要素を識別するためのものではない．また，ユーザが作成するモデルには，無名のノードや，同一の名前を持つノード群が存在しうるため，ノード名を識別文字として利用することは出来ない．

解析部は，入力の XMI ファイルから UML モデルを読み込んだ後に，*Activity* のノード名に一意な番号を付加して，UML モデルの内容を新たな別の XMI ファイルに出力する．*Activity* の中のノードの名前は，ノードごとの独立した情報であるため，UML モデル中の他の要素への影響を考慮せずに変更することができる．

新たに作成された XMI ファイルを入力として，表示部を実行すると，ノード番号を付加した Activity 図が表示される．ユーザは，表示内容を参照することで，解析結果中の識別文字とノードの対応を認識することができる．図 4.9 では，例題モデルの PC の操作 `WSExecutionAgent::sysExecuteWS` の解析結果と，ツールが自動で付与したノード番号付の Activity 図を示している[*3]．

本研究では，UML 処理系の MDT/UML2（MDT project [127]）を利用して，ツールの解析部の実装を行った．また表示部には，UML モデルの編集機能を有するツール Topcased（TOPCASED project [133]）を利用した．

[*3] 図 4.9 の Activity 図は，図 2.11 の振舞いモデルと同一の内容を示しているが，図 2.11 では，説明のために各ノードに手動で識別番号を付加しており，ノード番号の体系が異なる．

第5章

評価

本章では，3 章で導入した OCL 記述の解析手法と，4 章で導入した Activity の解析手法の評価を行う．5.1 節と 5.2 節で，OCL 記述の解析手法と Activity の解析手法のそれぞれについて，解析のアプローチの妥当性，解析能力，適用範囲，利用方法を議論する．5.3 節で，実システムを用いて行った評価実験について述べる．実験では，2 つの解析手法を情報システムの開発に適用して，実問題に対する手法の有用性を確認する．5.4 節で，関連研究について論じる．

5.1　議論：OCL 記述の解析手法

本節では，OCL 記述の解析方法の議論を行う．最初に 5.1.1 項で，本手法が想定する代入・取得・照合ノードの形式の妥当性を検討する．続いて 5.1.2–5.1.5 項で，4 つの技術課題に対する手法の解決アプローチ（表 3.1）の評価を行う．5.1.2 項では，技術課題 1（自動化・完全性・OCL の表現力の間のトレードオフ）に対応して，本解析の自動実行，近似の安全性，解析範囲，の 3 点について検討する．5.1.3 項では，技術課題 2（OCL の文脈依存性，非決定性，不完全性）に対応して，本手法の抽象解釈規則の厳密性を考察する．5.1.4 項では，技術課題 3（代入・取得・照合の識別）に対応して，解析全体の網羅性と正確性を把握する．5.1.5 項では，技術課題 4（照合情報の解析の計算量）に対応して，解析の効率性を評価する．5.1.6 項において，手法の適用範囲を述べる．

5.1.1　代入・取得・照合ノードの形式の妥当性

本手法では，代入ノード，取得ノード，照合ノードが，演算子 “=” を用いた等価関係，または演算子 “`includes`” 或いは “`includeAll`” を用いた *Collection* の包含関係の形式（代入・取得・照合ノードの形式）を持つことを仮定して，解析を行う．本項では，この形式の妥当性について検討する．

Cheesman et al. [47]，D’Souza et al. [50]，Ackermann et al. [75]，Cabot [77] に記載されている多くの事後条件の事例は，代入，取得，照合がいずれも上記の形式で記載されており，本手法の仮定が妥当であることを裏付けている．

Cabot [77] は，Frais et al. [129] や Queralt et al. [130] を含む，多数の OCL 記述の調査に基づいて，宣言的な OCL 記述を命令的なアクション記述に変換するパターンを提案した．Cabot [77] のパターンでは，事後条件中の宣言的な等価式 “`o.att=Y`” を，オブジェクト `o` の属性 `att` への `Y` の値の代入と解釈する．本手法の代入・取得・照合ノードの形式は，左右のオペランドの順序や形式に関する制約を持たず，Cabot [77] のパターンよりも多様な代入の記述をカバーする．

OCL の言語仕様では，本手法が仮定する “`=`”，“`includes`”，“`includeAll`”の 3 つの演算以外に，非等価比較の演算 “`<>`”，排除関係を表す演算 “`excludes`”，“`excludesAll`” を用いて，等価関係と包含関係を記述することが，原理的には可能である．例えば，等価関係 “`o.att = p`” と同等の内容を “`not o.att <> p`” と表現することが出来る．このような記述は冗長であるため，実際の事例では稀だと思われるが，本手法を次のように拡張することで，そのような形式に対応することが可能である．

- 代入・取得・照合ノードの形式に，演算 “`<>`”，“`excludes`”，“`excludesAll`” を用いた表現を追加する．
- Step 1 を実行する際に，演算 “`<>`”，“`excludes`”，“`excludesAll`” の OCL 表現ノード `E` が代入，取得を表すための意味的な制約として，条件 1_{AG} の代わりに “ノード `E` の値が `false` となる，事後条件を満たすオブジェクト状態が存在する” ことを課する．
- Step 2 を実行する際に，演算 “`<>`”，“`excludes`”，“`excludesAll`” の形式の照合ノード `E` に対して，“`not E`” の値を照合結果として扱う．

5.1.2 解析の自動実行・近似の安全性・解析範囲

2.5.1 項で述べたとおり，UML/OCL モデルの検証において，検証の自動化と完全性，モデルの表現力の 3 つの間にトレードオフが存在する（技術課題 1）．本論文では，抽象解釈技術を適用して，この技術課題の解決に取り組んだ．本項では，提案手法の解析について，実行の自動化，近似の安全性（本来の解釈結果を漏れなく含んでいること），解析対象の OCL 記述の範囲，の 3 つの観点から考察を行う．

まず，解析の自動実行について述べる．本手法の Step 1–2 の各解析は，いずれも機械的に実行することができる．また，後ほど 5.1.5 項で説明するように，事後条件の抽象構文木のノード数 N，分岐・合流点ノードの下層ブロックの組み合わせ数 N_c に対して，手法全体の時間計算量は $\mathrm{O}(N \times N_c)$ である．すなわち，本手法の解析のアルゴリズムは有限の計算時間で停止することが保証されており，その計算量はノード数 N の線形オーダーである[*1]．従って，本手法の一連の解析は自動で実行することができる．実際に 3.8 節で述べた支援ツールは，Step 1–2 の解析を完全に自動で実行し，解析結果を出力する．

[*1] ただし，N_c は分岐点と合流点の合計ノード数の指数オーダーで増大する．

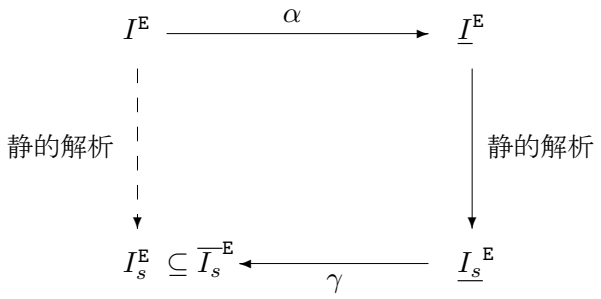


図 5.1. 具体領域と抽象領域におけるノード E の解釈規則

次に，抽象解釈の近似の安全性について考察する．2.5.1 項で述べたとおり，OCL 記述の制約を満たすオブジェクト状態は無限に存在しうる．抽象解釈を用いた解析結果の中に，とりうるオブジェクト状態に対する本来の解釈が漏れなく含まれる，安全な近似であることが望ましい．

付録 A に記載した 57 の解釈規則の定義方法は，以下の 2 つに大別される．

方法 1： OCL 表現の親子ノードが取り得る具体値の組を考えた後に，抽象化を行い，親子ノードが取り得る抽象値の組に変換する．

方法 2： OCL 表現の親子ノードが取り得る具体値の組を定義できない場合に，抽象領域上でのみ，親子ノードが取り得る抽象値の組を定める．

Collection に対する演算 `size`，`count`，`collect` と，それらを用いて定義される演算の *IteratorExp*/*OperationCallExp* ノードのバリエーションは，方法 2 によって解釈規則が定義され，その他の全てのバリエーションは，方法 1 によって解釈規則が定義される．

抽象構文木が，方法 1 のバリエーションのみで構成されている場合を考える．OCL 表現ノード E と $n_{\mathtt{E}}$ 個の子ノードが取り得る，具体値の組み合わせ集合と，抽象値の組み合わせ集合を，それぞれ $I^{\mathtt{E}} \subseteq (D - \{OclVoidValue\}) \times D^{n_{\mathtt{E}}}$，$\underline{I}^{\mathtt{E}} \subseteq (\underline{D} - U) \times \underline{D}^{n_{\mathtt{E}}}$ とする．$I^{\mathtt{E}}$ と $\underline{I}^{\mathtt{E}}$ は，ノード E の具体領域，抽象領域上の解釈規則を表す．方法 1 では，$I^{\mathtt{E}}$ の要素に抽象化を行うことによって，$\underline{I}^{\mathtt{E}}$ を求める．すなわち $\forall(d, d_1, .., d_{n_{\mathtt{E}}}) \in I^{\mathtt{E}}((\alpha(d), \alpha(d_1), .., \alpha(d_{n_{\mathtt{E}}})) \in \underline{I}^{\mathtt{E}})$ が成り立つ．$I^{\mathtt{E}}$ と $\underline{I}^{\mathtt{E}}$ の関係を図 5.1 に示す．

3.2.3 項で説明した静的解析は，他ノードの値との組み合わせによって事後条件が成立し得るような，各ノードの解釈規則の部分集合を求める．本手法では，静的解析を抽象領域で実行して $\underline{I_s}^{\mathtt{E}} \subseteq \underline{I}^{\mathtt{E}}$ を求めるが，同様の解析を具体領域で実行して，具体値の組み合わせ集合 $I_s^{\mathtt{E}} \subseteq I^{\mathtt{E}}$ を求めることが可能である．もしノード E，E′，.. の具体値の組 $(d, d_1, .., d_{n_{\mathtt{E}}}) \in I_s^{\mathtt{E}}$，$(d', d'_1, .., d'_{n_{\mathtt{E}'}}) \in I_s^{\mathtt{E}'}$，.. の組み合わせが事後条件を満たすならば，対応する抽象値の組 $(\alpha(d), \alpha(d_1), .., \alpha(d_{n_{\mathtt{E}}})) \in \underline{I}^{\mathtt{E}})$，$(\alpha(d'), \alpha(d'_1), .., \alpha(d'_{n_{\mathtt{E}'}})) \in \underline{I}^{\mathtt{E}'})$，.. の組み合わせも事後条件を満たす．従って $\forall(d, d_1, .., d_n) \in I_s^{\mathtt{E}}((\alpha(d), \alpha(d_1), .., \alpha(d_n)) \in \underline{I_s}^{\mathtt{E}})$ が成り立つ．

3.2.1 項で導入した抽象領域と抽象化写像は，$\forall d \in D(d \in \gamma(\alpha(d)))$ を満たす．従って $\forall \overrightarrow{d} = (d, d_1, .., d_n) \in I_s^{\mathtt{E}}(\overrightarrow{d} \in (\gamma(\alpha(d)), \gamma(\alpha(d_1)), .., \gamma(\alpha(d_n)))$ が成り立つ．すなわち，抽象

領域上の解析結果 $\underline{I_s}^{\mathrm{E}}$ を具体化した $\overline{I_s}^{\mathrm{E}}$ は，具体領域上の解析結果 I_s^{E} を含む（図 5.1）．

以上の考察から，方法 1 のノードのみで構成された抽象構文木に対しては，本手法の解析が，具体領域上での解釈結果を漏れなく抽出する，安全性を備えていることが分かる．従って，そのような OCL 記述については，制約を満たすオブジェクト状態を漏れなくカバーする，完全な解析を実行する．しかし，方法 2 のノードについては，具体領域での解釈規則が定義できないため，方法 2 のノードを含む抽象構文木に対しては，近似の安全性が保証されない．すなわち，制約を満たす一部のオブジェクト状態の内容が，解析結果に反映されていない可能性がある．

続いて，本手法が解析対象とする OCL 記述の範囲について論じる．3.2.2 項で述べたとおり，本手法では，OCL メタモデルで定義されている 12 種類の OCL 表現のうち，BasicOCL パッケージに対応する 9 種類の OCL 表現を扱う（図 A.1）．残りの *StateExp*，*MessageExp*，*AssociationClassCallExp* の 3 種類は，それぞれ状態，メッセージ，関連クラスを扱う OCL 表現である．2.3.2 項で導入した構造モデルは，状態とメッセージを扱わないため，*StateExp*，*MessageExp* の解析は不要である．一方，関連クラスは，データ間の関係に付随する情報を定義するための UML のメタモデル要素であり，実問題の構造モデルを記述する上で有用な場合がある．今後，関連クラスに関する OCL 制約を扱うために，手法の拡張を進める予定である．

また，3.2.1 項で説明した抽象化写像 α では，*Collection* の抽象化を行う際に，*Collection* の空，非空の情報に着目して，*Collection* 要素の順番の情報を捨象する．従って，本手法では，要素の順番を持つ *Sequence* と *OrderedSet* を扱うことができない．しかし，これらの順序付きの *Collection* は，実問題の構造モデルを記述する上で，時に有用である．順序付きの *Collection* に関する OCL 記述の解析を行うためには，*Collection* 要素の順序の情報を捨象しない抽象化の方法が必要であり，今後の研究課題である．

5.1.3 抽象解釈規則の厳密性

本項では，OCL 記述の文脈依存性・非決定性・不完全性に関する，抽象解釈規則の厳密性について考察する．

抽象領域での解釈規則 $\underline{I}^{\mathrm{E}}$ は，OCL 言語仕様（OMG [27]）で規定されている意味論に基づいて設計されており，OCL の非決定性，不完全性を反映している．また $\mathrm{U} \notin dom(\underline{I}^{\mathrm{E}})$ であり，親ノードの値が不定値 U の場合に，子ノードの値を特定する過度な解釈を避け，不完全性を確保している．さらに 3.2.3 項のアルゴリズム `CALCULATE-ABSTRACT-VALUES` は，抽象構文木の根ノードから下向きに有効な解釈の組み合わせを求めることにより，各ノードの文脈に則した解釈を行っている．以上のことから，本手法の抽象解釈は，技術課題 3 で述べた OCL の性質（文脈依存性・非決定性・不完全性）を厳密に取扱っていると考えられる．

但し，本項の考察は，3.2.2 節で定義した抽象領域における OCL 解釈規則が，OCL の意味論に則したものであることを前提としている．もし解釈規則に誤りが含まれる場合には，検討結果にも誤りがある可能性がある．

5.1.4　代入・取得・照合の識別性能

本手法の代入情報，取得情報，照合情報の識別性能について，正確性と網羅性の観点から論じる．

まず，網羅性について述べる．Step 1 で代入ノード，取得ノード，照合ノードを，Step 2.1 で照合出力ノード候補を，それぞれ独立に求める上では，抽象解釈を用いた近似的な計算が有効に機能する．しかし，照合ノード $\mathtt{E_C}$ と照合出力ノード候補 $\mathtt{E_0}$ の組合せについて，Step 2.2 で対応 $f_{\mathtt{E_0},\mathtt{E_C}}$ を求めて，Step 2.3 で含意判定を行う際に，$\mathtt{E_0}$ の値の近似の精度不足による偽陰性のエラーが生じ得る．

例として，事後条件 “p=‘NG’ or (p=‘OK’ and exp)”を考える．p は出力パラメータ，exp は照合を表す記述である．具象領域では，p に該当する照合出力ノード候補 $\mathtt{E_0}$ の値が ‘OK’ の場合に限り，exp に該当する照合ノード $\mathtt{E_C}$ は値 true を取り，条件 6 を満たす．しかし本手法の抽象解釈では，具体値 ‘OK’ と ‘NG’ は共に抽象値 T に対応付けられるため，$dom(f_{\mathtt{E_0},\mathtt{E_C}}) = \{\mathrm{T}\}, f_{\mathtt{E_0},\mathtt{E_C}}(\mathrm{T})=\{\mathrm{T},\mathrm{U}\}$ となり，Step 2.3 の含意判定で $\mathtt{E_0}$ を照合出力ノードから除外するエラーが発生する．

上記の例のような，*String* 又は *Enumeration* 型の出力パラメータとリテラル値の等価関係は，実問題の OCL 記述でしばしば見られる．これらの型の具体値は，常に抽象値 T に対応付けられるため，近似の精度不足によるエラーの恐れがある．そこで本手法では，そのような形式に合致する照合出力ノード候補を抽出した場合は，Step 2.3 の含意判定を行わずに，照合出力ノードの決定をユーザに委ねる．

次に，手法の正確性について述べる．本手法では，3.1.3 項で定義した “代入・取得・照合ノードの形式”に合致する照合ノード毎に，照合情報を求める．この方法は，操作の実行を通じてオブジェクト状態が変化する場合に，偽陽性の照合を抽出する可能性がある．

例として，事後条件 “result=(x.att@pre=p and not x.att=p)”を考える．p と x は，照合値を指定する入力パラメータと照合先を表すオブジェクトであり，result は操作の戻りパラメータである．本手法は，等価関係 “x.att@pre=p”に該当する照合ノード $\mathtt{E_C}$ と，result に該当する照合出力ノード $\mathtt{E_0}$ から構成される照合を抽出する．しかし，事後条件中の表現 “not x.att=p”は，照合先の値の更新やオブジェクトの削除によって，操作の事後に照合データが存在しないことを表している．この種の誤抽出のエラーを防ぐためには手法の拡張が必要であり，今後の研究課題である．

最後に，手法の網羅性と正確性の両方に影響を与えうる，解析方法の 2 つの制限事項を述べる．第 1 に，本手法の抽象解釈では，事後条件の抽象構文木の構造の情報のみに基づいて，ノード値の計算を行う．この方法では，不変条件によってノードに課される制約が考慮されないため，データアクセスの識別の際に，偽陽性・偽陰性の両方の誤りが生じうる．誤りを防ぐための方法は，今後の研究課題である．第 2 に，事後条件の抽象構文木中に，同一の意味を表す複数の OCL 表現ノードが含まれるケースがあり得るが，本手法の抽象解釈では，それらのノード値の制約が考慮されていない．例えば，表現 “X->reject(Y)”と表現 “X->select(not

表 5.1. 抽象構文木の属性

名前	意味
N	ノード数
L	ノード間のリンク数
N_b	ブロック数
L_b	ブロック間のリンク数
N_c	分岐・合流点ノードの下層ブロックの組み合わせ数

表 5.2. 解析の計算量

Step	解析	時間計算量	空間計算量
1.1	抽象解釈	$\mathrm{O}((N_b+L_b)N_c)$	$\mathrm{O}((N_b+L_b)N_c)$
1.2	起源の解析	$\mathrm{O}(N+L)$	$\mathrm{O}(N)$
1.3	データ流向の特定	$\mathrm{O}(1)$	$\mathrm{O}(1)$
2.1	確定性の判定	$\mathrm{O}((N_b+L_b)N_c)$	$\mathrm{O}((N_b+L_b)N_c)$
2.2	値の対応付け	$\mathrm{O}((N_b+L_b)N_c)$	$\mathrm{O}((N_b+L_b)N_c)$
2.3	含意判定	$\mathrm{O}(1)$	$\mathrm{O}(1)$

Y)”は互いに同一の意味を表しており，これらの表現が共に事後条件中に含まれる場合に，それらの OCL 表現ノードは同値をとる．しかし，本手法では，これらの表現を独立に解釈するため，誤ったノード値の組合せが出力される可能性がある．この誤りを防ぐための方法は，今後の研究課題である．

5.1.5 解析の効率性

本論文では，OCL 記述の解析手法の効率性を，次の 2 つの観点から評価する．

- 解析アルゴリズムの計算量
- 実問題の OCL 記述に対する実行性能

ここでは，前者の解析アルゴリズムの計算量について述べる．後者の実問題に対する実行性能については，5.3.1 節で議論する．

抽象構文木を特徴づける属性を表 5.1 に，各サブステップ毎の 1 事後条件あたりの解析の時間計算量と空間計算量を表 5.2 に，それぞれ示す[*2]．属性 L，N_b，L_b は何れも $\mathrm{O}(N)$ であり，手法全体の時間計算量，空間計算量は共に $\mathrm{O}(N \times N_c)$，即ちノード数 N の線形オーダーである．但し N_c は，分岐点と合流点の合計ノード数の指数オーダーで増大する．従って，複数箇所で参照される変数の数が少ない OCL 記述に関しては，本手法は技術課題 4 を解決している

[*2] 付録 B に計算量の見積り方法を記載する．

と言える．

本手法の各サブステップは，集合 S_c の各要素の計算を独立に実行することが出来る．Frias et al. [129] に記載されている事例のような，N が数百に及ぶ大規模な事後条件に対しても，計算の並列化によって実用的な計算時間で解析を行うことが可能である．

5.1.6　適用範囲

提案手法が扱う DC の振舞いモデルと構造モデルは，Catalysis（D'Souza et al. [50]）や UML Components（Cheesman et al. [47]）の CBD 方法論のコンポーネントモデルに適合する．これらの開発手法を採用する多数の実システムの開発で，本手法を適用することが出来る．大規模な構造モデルを扱うコンポーネントの照合操作では，仕様の記述と理解の誤りが起きやすい．そこで本論文では，コンポーネントベース開発を題材として手法の提案を行った．しかし，手法の解釈規則や解析方法そのものは，一般的な UML クラスの操作に対しても適用可能である．

3.1 節で述べたように，本手法では解析の対象を，単一の出力パラメータによって照合結果を伝える照合操作に限定している．また 3.7 節で述べたように，Step 2.3 の含意判定において，照合値が T になる場合に，出力ノードの抽象値が T，F の何れか一方に定まるという制限を設けている．これらの限定と制限は，実問題の照合操作の性質を想定して設けており，D'Souza et al. [50] と Cheesman et al. [47] に記載されている照合操作の事例に対して，例外なく適合することを確認した．さらに，他の多くの実システムの照合操作にも適合することが期待できる．本手法を拡張し，上記の限定と制限の範囲外のケースを扱うことは可能であるが，照合操作を識別する条件が複雑になり，解析の精度が低下する恐れがある．

本論文では述べなかったが，“`result=(X.att@pre=p or Y.att@pre=p)`” のように，複数の照合ノードの値の組み合わせによって照合結果が決定されるケースがありうる．本手法の Step 2.2，Step 2.3 を拡張することにより，このようなケースを取り扱うことが出来る．

5.2　議論：Activity の解析手法

本節では，Activity の解析手法について議論を行う．5.2.1 節で，技術課題 5 に対応して，破壊的アクセスと補完的アクセスを用いることの妥当性を検討する．5.2.2 節で，技術課題 6 に対応して，データフロー解析と整合性判定の網羅性と正確性を調べる．5.2.3 節で，技術課題 7 に対応して，整合性判定の網羅性と正確性を調べる．5.2.4 節で，判定結果の有効な利用方法を述べる．

5.2.1　破壊的アクセス・補完的アクセスの妥当性

本手法では，DC のデータに対するアクセス順序の判断基準を設けるために，Planas et al. [107, 108] の破壊的 *Action* と補完 *Action* のアイデアを複数の DC 間に拡張して，破壊的アクセスと補完的アクセスの組を定義し，破壊的アクセスと補完的アクセスの順序を定

めた．

DC 間の参照は，属性値を介した間接的な参照関係である点が，関係データベースの表間の参照に類似している．本手法では，この類似性に着目して，関係データベースにおける参照整合性を確保するために一般的に用いられる，次の 2 つの仕組みの類比として，破壊的アクセスと補完的アクセスを定義した．

- 参照制約動作（RESTRICT，CASCADE）[*3]
- データベース・アプリケーションのコード中の処理シーケンス

これら 2 つの方法は，関係データベース技術の多数の研究と実例を通じて確立されており，DC 間の参照整合性の問題に対しても自然に適合する．従って，データアクセス順序の判断基準を設けるために，破壊的アクセス・補完的アクセスを用いる本手法のアプローチは，妥当であると言える．

5.2.2 データフロー解析の性能

4.4 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，4.4.2 項の到達定義 *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* 間に跨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可能である．ただし，2.5.4 項のデータフロー解析の説明で述べたとおり，文脈依存性は解析の複雑さとトレードオフの関係にあるため，context-sensitive な解析方法では計算の複雑さが増大する．

5.2.3 整合性判定の性能

4.5 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無い．

[*3] 2.5.5 項で述べたように，その他の参照制約動作として，NO ACTION，SET NULL，SET DEFAULT が存在するが，これらはロールバックや，外部キーへの固定値の設定など，参照不整合が発生しうることを前提とした仕組みである．従って，本手法ではこれらの参照制約動作の類比を考慮しない．

4.5 節の整合性検証のアルゴリズム VERIFY は，次の 3 種類の欠陥を検出することができる．

欠陥 1： 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 2： 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 3： 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

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，判定結果に偽陰性と偽陽性の両方の誤りが生じ得る．前者の場合，VERIFY の出力結果が判定成功であっても，*Activity* の組み合わせによって参照整合性が破壊される．後者の場合は，VERIFY の出力結果が判定失敗であっても，実際には参照整合性が保たれる．

判定結果に偽陽性の誤りが生じる合成モデルの例を説明する．或る *Activity* の中で，先ず補完的アクセス $t'.e \in G_r^{dst}$ 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に基づく解析では，分岐条件の評価を行わない為，この同値関係を検出できず，アルゴリズムは誤って判定失敗と出力する．4.5.2 項で述べた $\mathtt{r}_2$ に関する判定失敗のケースは，この偽陽性の誤りに該当する．

5.2.4　判定結果の利用方法

判定結果の利用方法について述べる．本手法が判定失敗を出力した場合に，判定の対象である *Activity* の組み合わせに欠陥 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繋がるケースがあり得る．従って *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出来る．その際に，判定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，本手法が判定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為には，本手法の判定対象である *Action* の記述に加えて，制御フローの判定が必要である．例えば図 2.15 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，本手法では判定することが出来ない．また前述のように，本手法の判定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．

5.3 評価実験

実システムのコンポーネントベース開発において，OCL 記述の解析手法と Activity の解析手法を適用する評価実験を行った．

対象システム

実験の対象システムは，筆者の勤務先の社内システムであり，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 PC から構成される．5 つの DC には，合計 58 の操作と 61 の事後条件が定義される．DC 間には 25 の参照が存在する．即ち $|REF| = 25$ である．7 つの PC 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，各操作の振る舞いが *Activity* として記述される．

対象システムの開発は，次の特徴を有する．

- DC 間の参照整合性を確保しながら，多数のシステム機能を確実に実現する必要がある．
- 対象システムのソースコード（Java 言語）は 10 万行以上の規模を持つ．このような規模のシステムの整合性を人手により判定する場合に，漏れや誤りが発生する恐れがある．
- 複数の拠点で分散開発を行っており，開発の手戻りが発生した場合の費用・工数の面での影響が大きい．

上述の特徴は，手法の有用性を確認する上で好適であるため，評価実験の対象として選定した．

実験のプロセス

図 5.2 に示すプロセスに従って，実験を行った．このプロセスでは，次の各工程を順に実施して仕様モデルの定義を行う．

手順 1：DC 仕様定義

DC の構造モデルと，データアクセス操作の振舞いモデル（OCL）を定義して，仕様レビューを通じてモデルの洗練を行う．操作の事後条件に含まれる代入，取得，削除，照合を手動で抽出し，それらを特定するアクセス情報 AC を明示的に指定する．

手順 2：OCL 記述の解析

DC の振舞いモデルに対して，3.8 節で述べた支援ツールを実行して，代入情報，取得情報，照合情報を自動抽出する．手順 1 で手動で抽出したアクセス情報 AC と，自動抽出した代入情報，取得情報，照合情報を用いて，整合性 1 の判定を行う．不整合が存在する場合には，手順 1 の工程に戻って DC の仕様モデルの修正を行う．

手順 3 : PC 仕様定義／モデルの合成

DC の仕様モデルとアクセス情報 AC に基づいて，PC の構造モデルと，操作の振舞いモデル（Activity）を定義して，仕様レビューを通じて Activity 記述の洗練を行う．さらに，インテグレータが DC と PC の仕様モデルの合成を行い，合成モデルに対して仕様レビューを行う．この段階で，DC の振舞いモデルに関する不整合が新たに発見された場合には，手順 1 の工程に戻って DC の仕様モデルの修正を行う．

手順 4 : Activity の解析

インテグレータが，PC の仕様モデルに対して，4.6 節で述べた支援ツールを実行して，DC の操作の利用手順に関する整合性 2 の判定を行う．不整合が存在する場合には，手順 3 の工程に戻って PC の仕様モデルの修正を行う．

これらの仕様定義の工程に引き続き，コンポーネントの実装，合成，テスト，配備を行う実装レベルの開発フェーズへと進む．実装以降のフェーズで，DC，PC の仕様モデルの欠陥が発見された場合には，手順 1，手順 3 の工程に戻って，仕様モデルの修正を行う．

図 5.2 の仕様定義プロセスでは，手順 1 と手順 3 の仕様定義の直後に，手順 2 と手順 4 のツールを用いた整合性判定を実施する．従って，2.4.5 項で述べた，人手による検査に頼って仕様モデルの整合性を確保する開発プロセス（図 2.17）に比べて，図 5.2 の矢印 (A)–(C) で示した工程の手戻りの要因となる，仕様モデルの不整合の見落としの低減が期待できる．

以下では，5.3.1 項と 5.3.2 項で，OCL 記述の解析手法と Activity 解析手法のそれぞれについて，適用結果を説明し，手法の有用性を検討する．そして 5.3.3 項で，提案手法の汎用性と適用方法，DC に跨る整合性制約について考察を行い，評価実験をまとめる．

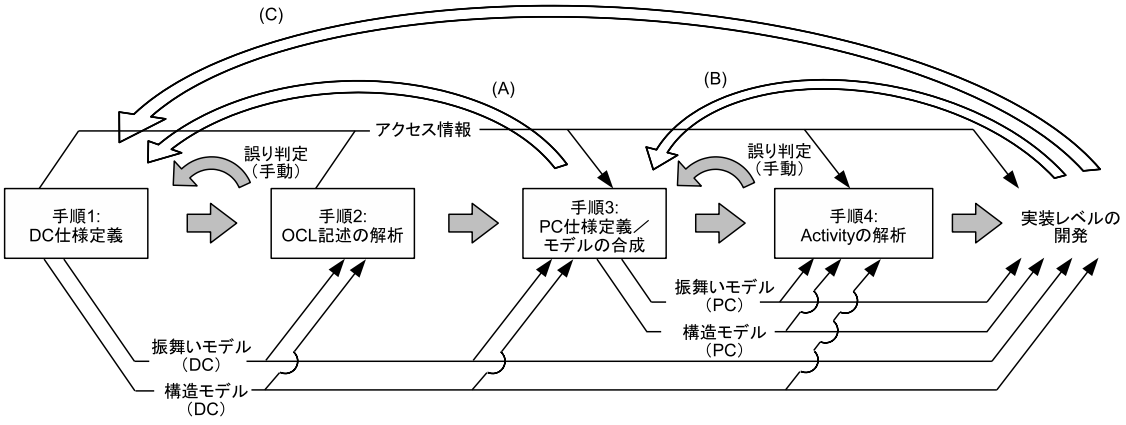


図 5.2. 提案手法を用いた評価実験のプロセス

表 5.3. OCL 表現ノードの出現頻度（百分率）

OCL 表現	ノード名	出現頻度	OCL 表現	ノード名	出現頻度
LiteralExp	リテラル	3.49	*OperationCallExp*	-	0.04
IteratorExp	exists	1.04		=	9.40
	forAll	1.21		allInstances	2.24
	isUnique	0.04		and	5.82
	any	1.16		asSet	1.04
	one	0.73		excludes	0.39
	collect	2.67		excludesAll	0.13
	select	2.50		implies	0.95
IfExp		0.04		includes	0.86
LetExp		1.51		includesAll	0.09
VariableExp	変数名	26.14		including	0.04
TypeExp	型名	2.24		isEmpty	0.35
PropertyCallExp		17.90		not	0.13
Variable		17.42		notEmpty	0.04
				or	0.17
				size	0.13
				union	0.04

5.3.1 OCL 記述の解析手法の有用性

実験の手順 1，2 には，DC の仕様記述者 2 名が被験者として参加した．被験者 1 は OCL の詳細な知識を有し，実開発で OCL を使用した経験を持つ．被験者 2 は OCL の基本知識を有し，実開発で OCL を使用するのは初めてである．

手順 1 の DC の振舞いモデル（OCL）の記述作業では，各被験者が異なる操作の記述を担当したが，それ以降の，モデルの洗練とアクセス情報の抽出の作業は，2 名が共同で実施した．2 名の被験者が分担して記述した 61 の事後条件の質は，共同の洗練作業を通じて均一化されたと見なせる．またアクセス情報の抽出は共同で作業を行っており，2 名の属人性は排除されている．手順 2 のツール実行と整合性 1 の判定は，人に依存しない機械的な作業である．従って，本実験の結果を元に手法の有用性を議論する際に，2 名の被験者の作業分担と，OCL の知識・経験の差異による影響を考慮しない．

手順 1 で作成された DC の振舞いモデルの事後条件に含まれる OCL 表現ノードの出現頻度を表 5.3 に示す．

以下では，実験結果に基づいて，OCL 記述の解析手法の評価を行う．代入・取得・照合の識別性能と，支援ツールの実行性能の 2 つの観点から，手法の有用性を検討する．

表 5.4. 実験で抽出したアクセス情報

	代入情報	取得情報	削除情報	照合情報
手動	$\|A_M\| = 49$	$\|G_M\| = 91$	$\|D_M\| = 6$	$\|C_M\| = 44$
自動	$\|A_T\| = 59$	$\|G_T\| = 101$	N/A	$\|C_T\| = 35$
最終値	$\|A_F\| = 59$	$\|G_F\| = 100$	$\|D_F\| = 6$	$\|C_F\| = 44$
共通要素	$\|A_M \cap A_T\| = 42$, $\|A_M \cap A_F\| = 49$, $\|A_T \cap A_F\| = 52$	$\|G_M \cap G_T\| = 91$, $\|G_M \cap G_F\| = 91$, $\|G_T \cap G_F\| = 100$	$\|D_M \cap D_F\| = 6$	$\|C_M \cap C_T\| = 31$, $\|C_M \cap C_F\| = 44$, $\|C_T \cap C_F\| = 31$

代入・取得・照合の識別性能

手順 1 で被験者が手動で求めたアクセス情報を $AC_M = (A_M, G_M, D_M, C_M)$，手順 2 でツールを用いて自動抽出した代入情報，取得情報，照合情報を A_T，G_T，C_T，実装レベルの開発を経て最終的に確定したアクセス情報を $AC_F = (A_F, G_F, D_F, C_F)$ と呼ぶ．これらの情報の関係を表 5.4 に示す[*4]．

A_M，A_T，A_F，G_M，G_T，G_F の内容について調べたところ，次の結果が得られた．

- $A_M \cap A_T$ の 42 要素は，いずれも正しく抽出された代入であり，A_F にも含まれる．
- $A_M \setminus A_T$ の 7 要素は，実際には代入ではない，人手の作業における誤答であった．これらの 7 要素は，モデル記述者の意図に合わせて構造モデルまたは振舞いモデルを修正することによって，A_F の要素として確定された．
- $A_T \setminus A_M$ の 17 要素は，いずれもツールによって正しく抽出された取得であり，人手の作業における抽出漏れであった．この中の 10 要素は，モデル記述者の解釈を修正することによって，A_F の要素として確定された．残りの 7 要素は，モデル記述者の意図に合わせて構造モデルまたは振舞いモデルの修正を行ったため，A_F には含まれない．
- $G_M \cap G_T$ の 91 要素は，いずれも正しく抽出された取得であり，G_F にも含まれる．
- $G_T \setminus G_M$ の 10 要素は，いずれもツールによって正しく抽出された取得であり，人手の作業における抽出漏れであった．この中の 9 要素は，モデル記述者の解釈を修正することによって，最終的に G_F の要素として確定された．残りの 1 要素は，モデル記述者の意図に合わせて構造モデルまたは振舞いモデルの修正を行ったため，G_F には含まれない．

これらのデータから，支援ツールは 100 % の精度と再現率で，代入と取得を自動抽出したことが分かる．上記の，仕様記述者の誤答と抽出漏れは，次の 2 通りのデータ流向に関する取違えによるものであった．

[*4] 表中の N/A は，該当項目が存在しないことを表す．

@pre の取違え： @pre の有無を取違えて，意図と異なる流向を表す OCL 表現の記述を行うケース．

操作パラメータの方向の取違え： DC のデータアクセス操作のパラメータの方向（in/out）を取違えて，意図と異なる流向を表す構造モデル（UML）の記述を行うケース．

同様に，C_M，C_T，C_F の内容について調べたところ，次の結果が得られた．

- $C_M \cap C_T$ の 31 要素は，いずれも正しく抽出された照合であり，C_F にも含まれる．
- $C_M \setminus C_T$ の 13 要素は，実際には照合ではない，人手の作業における誤答であった．これらの 13 要素は，モデル記述者の意図に合わせて振舞いモデルを修正することによって，C_F の要素として確定された．
- $C_T \setminus C_M$ の 4 要素には，5.1.4 項で述べた，ユーザへの判断委譲と，オブジェクト状態の変化による誤抽出のエラーが，2 件ずつ含まれていた．前者の 2 要素は，人の判断によって C_F の要素として確定された．後者の 2 要素は，C_F には含まれない．

これらのデータから，支援ツールは約 89% の精度と 100% の再現率で照合情報を自動抽出したことが分かる．仕様記述者による 13 の誤答（$C_M \setminus C_T$）には，次の 2 通りの論理の誤謬が見られた．ここで X は照合データの集合を表す OCL 表現とする．

全称記号の不完全性の誤謬： 事後条件 “X->forAll(i|result->exists(..))” を，X が空ならば result も空と誤って解釈し，result を照合出力情報とするケース．正しくは，X が空の時に result は未定義値を取る．例題の getAccessibleWS の事後条件（図 2.8）は，実験対象システムの振舞いモデルを引用したものであり，上述した形式の表現が含まれる（5–6 行目）．実験の手順 2 において，不完全性の誤謬が検出されたため，モデル記述者の意図に合わせて，X が空の時に result が空となるように 4 行目の記述を追加する修正を行った．

前件否定の誤謬： 事後条件 “result implies not X->notEmpty()” の裏の命題 “not result implies X->notEmpty()” が成り立つという誤った解釈から，result を照合出力情報とするケース．正しくは，前件が不成立の場合には，後件中の X は未定義値を取る．

以上の実験結果から，OCL の解析手法の代入・取得・照合の識別性能についてまとめる．手順 1 の人手の作業において，データ流向の取違えによる代入情報と取得情報の誤答と抽出漏れが発生し，論理の誤謬による照合情報の誤答が発生した．これらの誤答と抽出漏れは，直後の手順 2 において，ツールを用いた自動解析によって同定され，後工程へ伝播する前に検出・除去された．すなわち，OCL の解析手法の適用によって，図 5.2 の矢印 (A)，(C) の手戻りを防止できたことになる．

手順 2 のツールの自動解析では，オブジェクト状態の変化による照合情報の誤答が発生した．本実験の場合には，人手による正しい作業結果と比較することによって，この誤答は即座

表 5.5. 実モデルに対する実行性能（OCL 記述の解析）

	平均値	最大値
N	38	88
N_c	2,748	87,480
解析時間 (s)	1.74	48.16

に同定された．しかし，人手の作業でも同一の誤った解釈が行われた場合には，誤答に気づかず，不整合が後工程へ伝播していた可能性がある．

今回の実験のように，OCL の使用経験を持つ仕様記述者であっても，人手で作業を行う場合には，流向の取違えや，非決定性・不完全性の誤謬が起こる恐れがある．仕様記述者が操作仕様の作成時に，OCL 記述の解析手法を用いて誤謬を取り除くことによって，正確な仕様を記述することができる．さらに，DC のデータアクセス操作の利用者が操作仕様を正しく理解するうえでも同様に，手法を活用することができる．

支援ツールの実行性能

実験において，対象モデルの属性 N，N_c と，支援ツールの実行時間を計測した．N は事後条件の抽象構文木のノード数，N_c は抽象構文木の分岐・合流点ノードの下層ブロックの組み合わせ数である．計測には，一般的な能力 （Intel Core2 Duo 1.60GHz，4GB RAM）を持つ PC を使用した．計測結果を表 5.5 に示す．

3 章で説明したように，OCL 記述の解析における Step 1 の計算（抽象解釈，起源の解析，データ流向の特定）は，代入ノード，取得ノード，照合ノードを抽出するための共通処理として実施される．また，Step 2.1–2.2 の計算（確定性の判定，値の対応付け）では，Step 1 の抽象解釈で求めた S_c の計算結果を利用する．従って，代入・取得・照合の解析時間を個別に分離して扱うことはできないため，表 5.5 では，Step 1 と Step 2 の解析の合計時間を示している．

ツールは，平均的な N と N_c を持つ事後条件の解析を 2 秒未満で実行する．この実行時間は，人間がモデルの記述や理解に要する時間に比べて明らかに短く，ストレスなくツールを使用することが出来る．従って，OCL 記述の解析手法は，本実験の平均的な N と N_c と同程度の規模と複雑度を持つ事後条件を解析する上で，実用上十分な効率を備えていると言える．

5.3.2　Activity 解析手法の有用性

実験の手順 3，4 には，PC の仕様記述者数名と，仕様レビューア 3 名が被験者として参加した[*5]．被験者はいずれも UML Activity の知識を有し，実開発で UML を使用した経験を持つ．手順 3 の *Activity* 記述の仕様レビューの作業は，レビューア 3 名が共同で実施した．48 の *Activity* 集合の質は，洗練作業を通じて均一化されたと見なせる．従って，本実験の結果

[*5] PC の仕様記述を社外の開発委託先で実施したため，仕様記述者の正確な人数を把握することが出来なかった．

表 5.6. *Activity* 集合の規模

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

表 5.7. *Activity* ノードの出現頻度（百分率）

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 5.8. ツールを用いた自動判定の結果

成功	失敗				
27	8	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
		5	1	1	4

を元に手法の有用性を議論する際に，仕様記述者と，レビューア 3 名の被験者の作業分担と，*Activity* に関する知識・経験の差異による影響を考慮しない．

手順 3 の入力として用いる DC の構造モデルと振舞いモデル，アクセス情報 AC は，手順 2 の結果を元にして不整合を除去したものであり，AC は，実装レベルの開発を経て最終的に確定した AC_F と等しい． 手順 3 で作成された *Activity* 集合の規模と，*Activity* のノード種別毎の出現頻度を表 5.6，表 5.7 に示す．手順 4 において，手順 2 と同一の PC（Intel Core2 Duo 1.60GHz，4GB RAM）を用いて自動解析を行ったところ，ツールの実行時間は 833ms であった．

以下では，実験結果に基づいて，Activity 解析手法の評価を行う．*Activity* の整合性判定の性能の観点から，手法の有用性を考察する．

整合性判定の性能

手順 4 の自動判定の結果を，表 5.8 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 操作を成功，8 操作を失敗と判定した．判定失敗のケースを詳細に調べたところ，7 個のモデルの欠陥と，4 個の判定誤りが含まれていた．ただし，幾つかの判定失

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，5.2.3 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，何れも 5.2.3 節で述べた，制御フローに基づく偽陽性の誤りであった．これらの判定誤りは，支援ツールの出力する解析情報を用いて同定された．

手順 4 の Activity 解析手法の適用により，手順 3 の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出来た．これらの欠陥は，ツールを用いた自動解析によって同定され，後工程へ伝播する前に検出・除去された．すなわち，Activity 解析手法の適用によって，図 5.2 の矢印 (B) の手戻りを防止できたことになる．

7 個の欠陥の具体的な記述内容は欠陥毎に異なるが，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 5.6）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，複雑な操作の呼び出し関係により，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* に跨る確認を行う場合において，特に有用である．

また，5 つのシステム操作の整合性判定には，Planas et al. [107, 108] 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 5–7 を解決することで，これらの判定と欠陥の発見が可能となった．

本実験を通じて，技術課題 5–7 の実用上の重要性と，Activity の解析手法の有用性を確認することができた．提案手法を用いて，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．

5.3.3　評価実験のまとめ

本項では，提案手法の汎用性と適用手順，DC に跨る整合性制約について考察を行い，評価実験をまとめる．

手法の汎用性

本実験では，表 5.3，表 5.7 の構成要素と，表 5.5，表 5.6 の規模をもつ DC の振舞いモデル，PC の振舞いモデルを用いて，提案手法の有用性を確認した．

本研究で提案した OCL 記述の解析と Activity の解析の 2 つの解析手法は，いずれも標準的な UML/OCL で記述された仕様モデルを解析するアプローチを採っており，言語標準に従うモデルに対して汎用的に適用することができる．従って，本実験の対象システム以外の，他の多くの実システムに対しても同様に提案手法を適用して，仕様定義段階で DC 間の参照整合性を確保することが可能である．

手法の適用手順

本実験のプロセス（図 5.2）では，手順 1，手順 2 で DC のデータアクセスの整合性（整合性 1）の判定を行った後に，手順 3，手順 4 で PC の振舞いにおける DC の操作の利用手順の整合性（整合性 2）の判定を行った．その際に，手順 1（手順 3）の人手による判定の後に，手

順 2（手順 4）で支援ツールを用いて自動判定を実施した．

一方，本実験とは異なる以下の手順で解析手法を適用するプロセスも考えられる．

- 本実験の手順 1（手順 3）の仕様レビューと，手順 2（手順 4）の実施順序を逆にして，支援ツールを用いて自動判定を実行した後に，仕様レビューを行う．この場合には，ツールの解析結果を参照しながらレビューを行うことによって，人手による判定の正確性と効率の向上が期待できる．
- DC と PC が異なるサイクルで開発され，整合性 1，整合性 2 を独立して確保する必要があるような状況において，OCL 記述の解析または Activity の解析の何れか一方を単独で実施する．2 つの解析手法は，互いに依存関係を持たないため，単独で実施した場合にも，それぞれ整合性 1，整合性 2 を正確かつ効率的に判定する上で有効に機能することが期待できる．

これらのプロセスに基づく評価を実施して，提案手法の有用性と改善点を確認することは，今後の研究課題である．

DC に跨る整合性制約

2.3.2 項において，DC のデータモデルの整合性制約について概観し，Costal et al. [57] による整合性制約の分類の定義を述べた．同一 DC 内の異なる *Class* 間の関連のナビゲーションを含む整合性制約は，PathComparison，ValueComparison の何れかに分類される．ここで，DC 間の参照 $r = (cls_{src}, prop_{src}, cls_{dst}, prop_{dst}) \in REF$ を，cls_{src} と cls_{dst} の間の仮想的な関連と見做して，r のナビゲーションを含む PathComparison，ValueComparison の整合性制約を定義することが可能である．そして，そのような DC に跨る整合性制約は，本研究で扱った DC 間の参照整合性のケースと同様に，DC，PC の操作の振舞いモデルの組合せによって実現される．

評価実験において実際に，集合 *REF* を元にして，DC に跨る PathComparison，ValueComparison の整合性制約の集合 *IC* が定義された．実験では，図 5.2 の手順 1 において，非形式的な方法で *IC* を記述し[*6]，手順 1，手順 3 の仕様レビューにおいて，人手によって，*IC* に関する DC，PC の振舞いモデルの整合性判定を行った．その際に，判定の漏れや誤りが生じ，その後の実装レベルの開発工程において，*IC* に関する振舞いモデルの不整合が表面化して，開発工程の手戻りが発生した．

本研究では，仕様定義段階で DC 間の参照整合性を確立するために，UML/OCL の振舞いモデルの静的解析手法を提案して，その有用性を確認した．今後，さらなる開発生産性の向上を目指すためには，他の種類の DC に跨る整合性制約に関する，仕様モデルの整合性を確保するための研究に取り組む必要がある．

[*6] OCL の言語標準の範囲では，DC 間の参照のナビゲーションを表現することができない．

5.4　関連研究

5.1.4 項で述べたように，事後条件の抽象構文木中に，同一の意味を表す複数の OCL 表現ノードが含まれるケースがあり得る．Cabot et al. [131] は，OCL の表現を，等価な意味を持つ他の表現に変換する方法を述べている．Cabot et al. [131] の方法を応用して，事後条件中の同一の意味を表す OCL 表現ノードを特定する方法を検討する価値がある．そして，特定されたノードが同値をとる制約の下で抽象構文木の解釈を行うように，提案手法の抽象解釈を拡張することによって，無効なノード値の組合せを除外できる可能性がある．

Cabot et al. [89] は，UML/OCL モデルに期待される典型的な性質として，充足可能性，非冗長性に関する次の 5 つの性質を挙げた．

Strong satisfiability： 全ての *Class* と関連のインスタンス集合が非空であるようなオブジェクト状態が存在すること．

Weak satisfiability： 少なくとも 1 つの *Class* のインスタンス集合が非空であるようなオブジェクト状態が存在すること．

Liveliness of a class X： クラス X のインスタンス集合が非空であるようなオブジェクト状態が存在すること．

Lack of constraint subsumption： 2 つの異なる OCL 制約 c_1，c_2 について，c_1 を満たし，かつ c_2 を満たさないオブジェクト状態が存在すること．この性質が成立しない場合は，c_1 に包含される冗長な制約として，c_2 を削除することができる．

Lack of constraint redundancy： 2 つの異なる OCL 制約 c_1，c_2 について，c_1 または c_2 の一方のみを満たすオブジェクト状態が存在すること．この性質が成立しない場合は，c_1 または c_2 の何れかを，冗長な制約として削除することができる．

本論文で提案した OCL 記述の抽象解釈の方法を応用して，上述の性質を満たすオブジェクト状態の存在の有無を，抽象領域上で判定することが可能である．2.5.1 項で説明したように，従来，UML/OCL モデルの検証の自動化，OCL の表現力，検証の完全性の間のトレードオフを扱うために，定理証明技術を利用する方法，局所解の探索を行う方法，OCL 記述を他の論理体系にマッピングして検証を行う方法，の 3 通りのアプローチの研究が行われてきた．3 章で述べた，OCL 記述に対する抽象解釈の適用は，実用的な計算時間内で，取りうるオブジェクト状態に対する完全な自動解析を実現する，新しいアプローチである．その他の，UML/OCL モデルの検証に要する計算量を削減するためのアプローチとして，Shaikh et al. [132] でスライシング技術を利用する方法が提案されている．

本論文では，*Activity* 中の，破壊的アクセスと補完的アクセスを表す *Action* の順序関係の判定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，プログラム中の順序制約の検証が挙げられる． Olender et al. [121, 122] 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取り得る状態を解析することによって，イベ

ント記述と制約の整合性を静的に検証する方法を提案した．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出来る．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判定する，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．

本論文では，UML の *Activity* を対象として，オブジェクトの属性値の伝搬を扱う field-sensitive なデータフロー解析の方法を提案した．ビジネスプロセス記述言語の BPEL を対象としたデータフロー解析では，BPEL の例外をハンドリングする仕組みである “dead path elimination”（Curbera et al. [134]）の扱いが課題となる．Kopp et al. [135] は，“dead path elimination” を考慮した BPEL 記述の field-sensitive なデータフロー解析方法を述べている．

C 言語等の型のキャストを含むプログラムのデータフロー解析では，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．Wilson et al. [114] は，この課題に対処するために，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 の *Activity* においても，*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．しかし，2.3.2 項で述べた構造モデルにはモデル要素間の継承関係が存在しないため，PC の振舞いモデルには *ReclassifyObjectAction* が含まれない．従って，本手法では，上記のキャストに起因する課題は生じない．

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 [136]．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 Activity の 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．

Ait-Sadoune et al. [137] は，BPEL 記述を，形式手法である Event-B のモデルに変換して，定理証明技術を用いてモデルの性質を検証する方法を提案した．この方法は，BPEL 記述のデータと制御の流れを同時に扱う能力を持つ．今後，本論文で取り組んだ，UML Activity のデータアクセス操作の利用手順の整合性判定の問題に対して，定理証明技術を利用する新たなアプローチによる解法を検討する価値がある．

第 6 章

結論

本章では，本研究の結論を述べる．6.1 節で研究の結果をまとめ，6.2 節で今後の課題と展望を述べる．

6.1 本研究のまとめ

情報システムのコンポーネントベース開発において，仕様定義段階でビジネスコンポーネントに跨る参照整合性を確立するために，UML/OCL 振舞いモデルの整合性の判定方法の研究に取り組んだ．以下に研究の結果を要約し，提案内容の意義を確認する．

OCL 記述の解析

本論文では，抽象解釈技術を用いて，OCL 記述の意味の近似的な解釈を行い，OCL 抽象構文木上のノード値の由来とデータの流向を特定することによって，代入・取得・照合のデータアクセスを識別する，OCL 記述の静的解析手法を提案した．これは，筆者の知る限りでは，OCL 記述に対する抽象解釈技術の適用を行った，最初の提案である．本手法の利用者は，識別された情報を用いて，データアクセス操作の振舞いモデルに対する自身の解釈の整合性を容易に判定することができる．

本手法は，OCL 記述の意味の解釈を行う際に，OCL の言語特性である文脈依存性・非決定性・不完全性を厳密に取扱う．従来の研究で提案されている，パターンに基づいて OCL の記述・解釈を行う方法では，これらの言語特性が考慮されず，モデルの意味を過度に限定して解釈する “overspecification” の恐れがあった．

本手法のアルゴリズムは，OCL 言語仕様のある範囲において，取りうるオブジェクト状態に対する完全な解析を，OCL 抽象構文木のノード数の線形オーダーの計算量で自動実行することができるため，大規模なシステムの仕様モデルの解析に広く用いる上で好適である．従来の，定理証明やモデル検査などの技術に基づく UML/OCL モデルの整合性検証の方法では，計算の完全性・実用的な計算時間・自動化の 3 つの要件を同時に実現することができなかった．

Activity の解析

本論文では，UML Activity で記述されたコンポーネントの振舞いモデルの組合せに対して，データアクセス操作の利用手順の整合性を判定する，Activity の静的解析手法を提案した．本手法の利用者は，手法が出力する判定結果の情報を用いて，*Activity* の欠陥を調べることができる．

本手法は，ビジネスコンポーネント間の参照整合性を破壊しうるデータアクセスと，破壊を防ぐために必要なデータアクセスの組を定義し，*Activity* の属性依存のデータフロー解析を実施する．これらは，UML Activity の組合せを対象として，関係データベースの参照整合性の実現方法と，データフロー解析の技術を応用したものであり，従来にない新しい応用の形態である．

提案手法は，*Activity* 中のデータアクセス操作の引数の到達定義と，操作呼び出しの履歴情報を利用する，データアクセス手順の整合性の判定アルゴリズムを導入した．このアルゴリズムは，*Activity* 中のデータの流れに依存する，複雑なデータアクセス手順を扱うことができる．これまでの研究で提案されている，*Activity* 中の処理手順の検証方法では，データの流れが考慮されていなかった．

支援ツールと評価実験

本研究では，OCL 記述の解析と Activity の解析のそれぞれについて，解析を自動化するツールの開発を行った．大規模な実システムの開発に手法を適用する上で，手間のかかる解析手続きを自動化する支援ツールを提供することは極めて重要である．OCL 記述の解析ツールの利用者は，一連の解析を自動で実行して，その結果が自身の解釈と異なる場合には，ツールが表示する情報を参照しながら，解釈の違いが生じた箇所を詳細に調べることができる．Activity の解析ツールの利用者は，一連の解析を自動で実行して，整合性の判定結果を得ることができる．判定失敗のケースでは，判定の根拠となった詳細情報を参照して，*Activity* の欠陥を調べることができる．

さらに，実システムの開発に手法を適用する評価実験を行い，手法の有用性を確認した．OCL 記述の解析手法の適用に関して，人手による作業では，データ流向の取違えと OCL の非決定性・不完全性の誤謬によって，データアクセスの識別の漏れや誤りが発生したが，支援ツールはそれらを正しく識別した．また，Activity の解析手法の適用に関して，人間が苦手とする，多数の *Activity* に跨る整合性判定を行う場合に，支援ツールを用いた自動判定が特に有用であることが分かった．

以上の結果から，本論文で提案した OCL 記述の解析手法と Activity の解析手法は，仕様定義段階でビジネスコンポーネント間の参照整合性を確立する上で，有望なアプローチであると言える．したがって，本研究では，データの一貫性が保たれた高品質の情報システムを効率的に構築するための，UML/OCL モデルの整合性判定技術について，ソフトウェア工学の分野における学術的な寄与が出来たと考えられる．

6.2　今後の課題と展望

提案手法の適用範囲の拡大，性能の向上，適用事例の拡充，他の問題への応用の 4 つの観点から，今後の課題と展望を述べる．

適用範囲の拡大

本論文で述べた OCL 記述の解析手法では，要素の順序を持つ *Sequence* と *OrderedSet* を扱うことができないが，これらの *Collection* 型は，実問題の構造モデルを記述する上で時に有用である．手法を拡張して，*Collection* 要素の順序を考慮した抽象解釈を行うためには，順序情報の精度と計算コストのトレードオフを扱う必要がある．この問題の解法として，OCL 記述中の *Collection* の利用形態に応じて，適応的に抽象化の精度を調節する方法が考えられる．

OCL にはオブジェクトの消滅を明示的に表現する演算子が存在しないため，本論文で提案した OCL 記述の解析手法のように，データアクセスを表す OCL 表現ノードの形式を仮定する方法では，OCL 記述から直接，削除情報 D を抽出することはできない．この問題の解法として，本手法の抽象解釈を利用して，OCL 論理式の充足可能性を調べることによって，ある $d = (op, cls, prop_{cls}, param, prop_{param}) \in D$ の有無を間接的に求める方法が考えられる．まず，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 の全てのインスタンスが，操作を通じて存在し続けることを表す条件と，事後条件の連言を構成する．そして，この新たな論理式の充足可能性を抽象領域上で解析する．論理式が充足不可能である場合は d が存在せず，充足可能な場合には d が存在しうると結論づけることができる．

性能の向上

5.1.4 項で述べたように，本論文の OCL 記述の解析手法では，操作の実行を通じてオブジェクト状態が変化する場合に，偽陽性の照合を抽出する可能性がある．また，不変条件によって事後条件の抽象構文木のノード値に課される制約を考慮しないため，偽陽性・偽陰性のデータアクセスの識別誤りが生じうる．これらの問題の解法として，事後条件の論理式に，着目する性質を調べるための条件を加えた上で，抽象解釈を用いて解析を行う方法が考えられる．前者のケースでは，操作を通じてオブジェクト状態が変化することを表す条件と，事後条件の連言を構成し，新たな論理式の充足可能性を調べることによって，偽陽性の照合を抽出する可能性を確認する．後者のケースでは，不変条件と事後条件の連言に対して解析を行う．

本論文で提案した OCL 記述の抽象解釈では，事後条件の抽象構文木中の，同一の意味を表す複数の OCL 表現ノードの値の制約が考慮されていないため，誤ったノード値の組合せが出力される可能性がある．この問題の解法として，事後条件中の同一の意味を表す OCL 表現ノードを予め抽出して，それらのノードが同値をとる制約の下で抽象構文木の解釈を行うように，提案手法を拡張する方法が考えられる．

本論文では，*Activity* 中のオブジェクトの到達定義を解析する，field-sensitive なデータフロー解析の方法を導入した．この方法は，context-insensitive/path-insensitive 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．今後，操作の呼び出し箇所の

文脈と制御の分岐条件を考慮する，context-sensitive/path-sensitive な方法に拡張することによって，解析の精度の向上が期待できる．

本論文で，*Activity* の整合性判定のために導入したアルゴリズム VERIFY は，データ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，判定結果に偽陰性と偽陽性の両方の誤りが生じ得る．この問題を解決するために，定理証明技術を利用して，データと制御の流れを同時に扱う能力を持つ，新たな判定方法を検討する価値がある．一般に，定理証明の実行の際には利用者の介入が必要であるが，分岐が含まれる *Activity* の組合せに限定して定理証明技術を適用し，他の *Activity* の組合わせには本論文の提案手法を適用する，ハイブリッドなアプローチを採ることで，人手による作業を最小限に留めることが可能である．

適用事例の拡充

本論文では，実システムを用いた評価実験を通じて提案手法の有用性を確認したが，より多くの実開発に手法の導入を行い，有用性の実証を行う必要がある．また，実際の適用を通じて，手法の改良点のヒントが得られる可能性がある．例えば，5.1.4 項で述べたように，本手法では照合出力ノードの解析において，実問題の OCL 記述でしばしば見られる，*String* 又は *Enumeration* 型の出力パラメータとリテラル値の等価関係のノードの扱いをユーザに委ねることによって，偽陰性の識別誤りを回避している．このような，実問題の OCL 記述の特徴に則した実用的な改良によって，手法の有用性の向上が期待できる．

提案手法の応用

OCL 制約の充足可能性や記述の冗長性の有無を判定する問題に対して，2.5.1 節で説明した定理証明器や制約ソルバを利用する方法が提案されている．本論文で提案した OCL 記述の抽象解釈の方法は，取りうるオブジェクト状態を網羅的に扱う，実用的な計算時間内での自動解析を実現するアプローチとして，このような問題に応用できる可能性がある．

本論文では，*Activity* の前方からの情報の伝搬を調べる前方解析を行って，*ObjectNode* の到達定義を求める方法を示した．この方法を，後方への伝搬を調べる後方解析に応用して，*ObjectNode* の値が使用される箇所を求める定義使用の解析を行うことが可能である．

本論文では，*Activity* 中のデータアクセス手順の整合性を判定する方法を示した．判定の根拠として用いられる，*Activity* 中のデータアクセス操作の引数の到達定義と，操作呼び出しの履歴情報を利用して，*Activity* の欠陥の修正作業の支援に手法を応用できる可能性がある．*Activity* 中でデータアクセス操作を呼び出すべき位置の候補を計算して，利用者に提示するなどの支援が考えられる．

本研究は，データの一貫性が保たれた高品質の情報システムを効率的に構築するための，UML/OCL モデルの整合性判定技術の研究であり，本論文を通して，提案手法が有望なアプローチであることを示した．しかし，開発効率と品質のさらなる向上を実現するためには，今後も上述した課題の解決に向けて研究を進めていく必要がある．

発表文献

学術論文

(1) 井上 拓, 本位田 真一. 参照整合性の検証のための UML Activity 解析方法. 情報処理学会論文誌, Vol.54 No.2, pp.774–786, 2013.

(2) 井上 拓, 本位田 真一. 照合操作の識別のための OCL 解析方法. 情報処理学会論文誌, Vol.54 No.3, pp.1165–1184, 2013.

国際会議（査読付き）

(3) Taku Inoue, Shinichi Honiden. A method for data-flow analysis of business components. In Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering (CBSE '11). pp.51–60, New York, NY, USA. 2011.

特許

(4) キヤノン株式会社. 情報処理装置及びその方法、プログラム. 特開 2013-3853 号. 2013-01-07.

(5) キヤノン株式会社. 情報処理装置及びその方法、プログラム. 特開 2013-145537 号. 2013-07-25.

参考文献

[1] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, Vol. 24, No. 2, pp. 131–183, June 1992.

[2] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *Software, IEEE*, Vol. 4, No. 2, pp. 41–49, 1987.

[3] Peter Herzum and Oliver Sims. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2000.

[4] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, Vol. 39, No. 2, February 2006.

[5] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, Vol. 30, No. 4, pp. 459–527, December 1998.

[6] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together (ACM Press).* Addison-Wesley Professional, 6 2001.

[7] J. Sametinger. *Software Engineering with Reusable Components.* Springer, Heidelberg, 1997.

[8] D. McIllroy. Mass produced software components. In P. Naur and B. Randall, editors, *Software Engineering*, pp. 138–155, 1968. Report on a Conference by the NATO Science Committee.

[9] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, Vol. 32, No. 4, pp. 548 –566, 1993.

[10] W.W. Agresti and W.M. Evanco. Projecting software defects from analyzing ada designs. *Software Engineering, IEEE Transactions on*, Vol. 18, No. 11, pp. 988 –997, nov 1992.

[11] William B. Frakes, Ted J. Biggerstaff, Ruben Prieto-Diaz, Kazuo Matsumura, and Wilhelm Schaefer. Software reuse: is it delivering? In *Proceedings of the 13th international conference on Software engineering*, ICSE '91, pp. 52–59, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[12] J.C. Browne, T. Lee, and J. Werth. Experimental evaluation of a reusability-oriented

parallel programming environment. *Software Engineering, IEEE Transactions on*, Vol. 16, No. 2, pp. 111 –120, feb 1990.

[13] P. Vitharana, H. Jain, and F. Zahedi. Strategy-based design of reusable business components. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, Vol. 34, No. 4, pp. 460 –474, nov. 2004.

[14] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantiék Plašil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What characterizes a (software) component? *Software - Concepts & Tools*, Vol. 19, pp. 49–56, 1998.

[15] F.Bachmann, L.Bass, C. Buhman, S.Comella-Dorda, F.Long, J.Robert, R.Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering. Technical report, Software Eng. Inst., Carnegie Mellon Univ., 2000. Technical Report CMU/SEI-2000-TR-008.

[16] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, Vol. 33, No. 10, pp. 709 –724, oct. 2007.

[17] Richard Monson-Haefel. *Enterprise Javabeans (Java Series)*. Oreilly & Associates Inc, 3 sub edition, 10 2001.

[18] Sanjiva Curbera, Francisco Leymann, Frank Storey, Tony Ferguson, and Donald F. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 1 edition, 3 2005.

[19] OMG. CORBA Component Model Specification OMG Available Specification Version 4.0, 04 2006. `http://www.omg.org/spec/CCM/4.0/PDF`.

[20] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pp. 16–, Washington, DC, USA, 1996. IEEE Computer Society.

[21] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, Vol. 26, No. 1, pp. 70–93, jan 2000.

[22] OMG. OMG Unified Modeling LanguageTM (OMG UML), Superstructure Version 2.2, 02 2009. `http://www.omg.org/spec/UML/2.2/Superstructure`.

[23] A. Beugnard, J.M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, Vol. 32, No. 7, pp. 38–45, jul 1999.

[24] B. Meyer. Applying 'design by contract'. *Computer*, Vol. 25, No. 10, pp. 40–51, oct. 1992.

[25] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, Vol. 31, No. 3, pp. 1–38, May 2006.

[26] R. Kramer. iContract – the JavaTM design by ContractTM tool. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pp. 295 –307, aug 1998.

[27] OMG. Object Constraint Language OMG Available Specification Version 2.0, 05 2006. `http://www.omg.org/spec/OCL/2.0/`.

[28] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling With Syntropy (Prentice Hall Object-Oriented Series)*. Prentice Hall, 11 1994.

[29] S. J. Goldsack, K. Lano, and E. Dürr. Invariants as design templates in object-based systems. In *Proceedings of the 1st Workshop on Component-Based Sys-tems. Zurich, Switzerland, 1997, in conjunction with European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[30] D. Johnson and H. Kilov. An approach to an rm-odp toolkit in z. In *Proceedings of the 1st Workshop on Component-Based Sys-tems. Zurich, Switzerland, 1997, in conjunction with European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[31] R. Allen and D. Douence, R.and Garlan. Specifying dynamism in software architectures. In *Proceedings of the 1st Workshop on Component-Based Sys-tems. Zurich, Switzerland, 1997, in conjunction with European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[32] C. Canal and J. Pimentel, E.and Troya. On the composition and extension of software components. In *Proceedings of the 1st Workshop on Component-Based Systems. Zurich, Switzerland, 1997, in conjunction with European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[33] P. Henderson. Formal models of process components. In *Proceedings of the 1st Workshop on Component-Based Systems. Zurich, Switzerland, 1997, in conjunction with European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[34] M. Lumpe, J Schneider, O Nierstrasz, and F. Achermann. Towards a formal composition language. In *Proceedings of the 1st Workshop on Component-Based Sys-tems. Zurich, Switzerland, 1997, in conjunction with European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[35] Zied Choukair and Antoine Beugnard. Real-time object oriented distributed processing with coremo. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technologys*, Vol. 1357 of *Lecture Notes in Computer Science*, pp. 265–269. Springer Berlin / Heidelberg, 1998.

[36] X. Franch. Systematic formulation of non-functional characteristics of software. In *Requirements Engineering, 1998. Proceedings. 1998 Third International Conference on*, pp. 174 –181, 6-10 1998.

[37] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pp. 169–180, New York, NY, USA, 1990. ACM.

[38] Ian Holland. Specifying reusable components using contracts. In Ole Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, Vol. 615 of *Lecture Notes in Computer Science*, pp. 287–308. Springer Berlin / Heidelberg, 1992.

[39] Edsger W. Dijkstra. *Selected writings on computing: a personal perspective.* Springer-Verlag New York, Inc., New York, NY, USA, 1982.

[40] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, Vol. 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer Berlin / Heidelberg, 1997.

[41] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *Software, IEEE*, Vol. 12, No. 6, pp. 17–26, nov 1995.

[42] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, p. 179, april 1995.

[43] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, Vol. 1, No. 4, pp. 355–398, October 1992.

[44] J. Bosch. Superimposition: a component adaptation technique. *Information and Software Technology*, Vol. 41, No. 5, pp. 257–273, 1999.

[45] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 374–384, may 2003.

[46] Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 537–546, New York, NY, USA, 2002. ACM.

[47] J. Cheesman and J. Daniels. *UML Components - a Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[48] Antonia Albani, Sven Overhage, and Dominik Birkmeier. Towards a systematic method for identifying business components. In *Proceedings of the 11th International*

Symposium on Component-Based Software Engineering, CBSE '08, pp. 262–277, Berlin, Heidelberg, 2008. Springer-Verlag.

[49] Xiaofei Xu Zhongjie Wang and Dechen Zhan. A survey of business component identifcation methods and related techniques. *Internation Journal of Information Technology.*, Vol. 2, No. 4, pp. 229–238, 2005.

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

[51] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, 1990.

[52] OMG. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.2, 02 2009. `http://www.omg.org/spec/UML/2.2/Infrastructure`.

[53] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1, 01 2011. `hhttp://www.omg.org/spec/QVT/1.1/`.

[54] L.C. Briand, Y. Labiche, M. Di Penta, and H. Yan-Bondoc. An experimental investigation of formality in uml-based development. *Software Engineering, IEEE Transactions on*, Vol. 31, No. 10, pp. 833–849, oct. 2005.

[55] Michael Wahler. *Using Patterns to Develop Consistent Design Constraints*. PhD thesis, ETH Zurich, Switzerland, February 2008. No. 17643.

[56] E Miliauskaitė and L Nemuraitė. Representation of integrity constraints in conceptual models. *Information Technology and Control*, Vol. 34(4), pp. 355–365, 2005.

[57] Dolors Costal, Cristina Gomez, Anna Queralt, Ruth Raventos, and Ernest Teniente. Facilitating the definition of general constraints in uml. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, Vol. 4199 of *Lecture Notes in Computer Science*, pp. 260–274. Springer Berlin / Heidelberg, 2006.

[58] Barry Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pp. 12–29, New York, NY, USA, 2006. ACM.

[59] B. Boehm and V.R. Basili. Top 10 list [software development]. *Computer*, Vol. 34, No. 1, pp. 135–137, jan 2001.

[60] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering: Fundamentals*, Vol. 1, pp. 24–29. World Scientific, 2001.

[61] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of uml model consistency management. *Information and Software Technology*, Vol. 51, No. 12, pp. 1631–1645, 2009.

[62] M. Usman, A. Nadeem, Tai hoon Kim, and Eun suk Cho. A survey of consistency

checking techniques for uml models. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pp. 57–62, dec. 2008.

[63] Gregor Engels, Jochem M. Küster, Reiko Heckel, and Luuk Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. *SIGSOFT Softw. Eng. Notes*, Vol. 26, No. 5, pp. 186–195, September 2001.

[64] Gregor Engels, Reiko Heckel, and JochenMalte Kuster. Rule-based specification of behavioral consistency based on the uml meta-model. Martin Gogolla and Cris Kobryn, editors, ≪ UML ≫ 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools, 第 2185 巻 of *Lecture Notes in Computer Science*, pp. 272–286. Springer Berlin Heidelberg, 2001.

[65] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and JeanLouis Sourrouille. Consistency problems in uml-based software development. In Nuno Jardim Nunes, Bran Selic, Alberto Rodrigues da Silva, and Ambrosio Toval Alvarez, editors, *UML Modeling Languages and Applications*, Vol. 3297 of *Lecture Notes in Computer Science*, pp. 1–12. Springer Berlin Heidelberg, 2005.

[66] Hongji Yang. *Software Evolution with UML and XML*. Idea Group Pub, 4 2005.

[67] Daniel L. Moody. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data & Knowledge Engineering*, Vol. 55, No. 3, pp. 243–276, 2005.

[68] R. Salay. *Using modeler intent in software engineering,*. PhD thesis, University of Toronto, 2010.

[69] Anthony Finkelsteiin, George Spanoudakis, and David Till. Managing interference. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pp. 172–174, New York, NY, USA, 1996. ACM.

[70] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, Vol. 33, No. 4, pp. 24–29, apr 2000.

[71] M.W. Bright, A.R. Hurson, and S.H. Pakzad. A taxonomy and current issues in multidatabase systems. *Computer*, Vol. 25, No. 3, pp. 50–60, march 1992.

[72] R. Ahmed, P. Desmedt, W. Du, W. Kent, M.A. Ketabchi, W.A. Litwin, A. Rafii, and M.-C. Shan. The pegasus heterogeneous multidatabase system. *Computer*, Vol. 24, No. 12, pp. 19–27, dec. 1991.

[73] Witold Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. *ACM Comput. Surv.*, Vol. 22, No. 3, pp. 267–293, September 1990.

[74] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998. (邦訳 :『UML/MDA のためのオブジェクト制約言

語 OCL 第 2 版』 竹村 司 訳, エスアイビーアクセス, 2004 年).

[75] Jorg Ackermann and Klaus Turowski. A library of ocl specification patterns for behavioral specification of software components. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, Vol. 4001 of *Lecture Notes in Computer Science*, pp. 255–269. Springer Berlin / Heidelberg, 2006.

[76] A. Correa, C. Werner, and M. Barros. Refactoring to improve the understandability of specifications written in object constraint language. *Software, IET*, Vol. 3, No. 2, pp. 69–90, april 2009.

[77] Jordi Cabot. From declarative to imperative uml/ocl operation specifications. In Christine Parent, Klaus-Dieter Schewe, Veda Storey, and Bernhard Thalheim, editors, *Conceptual Modeling - ER 2007*, Vol. 4801 of *Lecture Notes in Computer Science*, pp. 198–213. Springer Berlin / Heidelberg, 2007.

[78] Taku Inoue and Shinichi Honiden. A method for data-flow analysis of business components. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, pp. 51–60, New York, NY, USA, 2011. ACM.

[79] フィリップクルーシュテン. ラショナル統一プロセス入門 第 3 版 (ASCII Software Engineering Series). アスキー, 2004.

[80] K. Beck. Embracing change with extreme programming. *Computer*, Vol. 32, No. 10, pp. 70–77, oct 1999.

[81] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering, IEEE Transactions on*, Vol. 21, No. 10, pp. 785–798, oct 1995.

[82] Shane Sendall and Alfred Strohmeier. Using ocl and uml to specify system behavior. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL*, Vol. 2263 of *Lecture Notes in Computer Science*, pp. 406–409. Springer Berlin / Heidelberg, 2002.

[83] Piotr Kosiuczenko. Specification of invariability in ocl. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, Vol. 4199 of *Lecture Notes in Computer Science*, pp. 676–691. Springer Berlin / Heidelberg, 2006.

[84] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artificial Intelligence*, Vol. 168, pp. 70–118, 2005.

[85] B. Beckert, U. Keller, and P. H. Schmitt. Translating the object constraint language into first-order predicate logic. In *VERIFY, Workshop at Federated Logic Conferences (FLoC)*, pp. 113–123, 2002.

[86] AchimD. Brucker and Burkhart Wolff. Hol-ocl: A formal proof environment for uml/ocl. In JoseLuiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches*

to Software Engineering, Vol. 4961 of *Lecture Notes in Computer Science*, pp. 97–100. Springer Berlin Heidelberg, 2008.

[87] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, Berlin, Heidelberg, 2007.

[88] Jordi Cabot, Robert Clariso, and Daniel Riera. Verifying uml/ocl operation contracts. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods*, Vol. 5423 of *Lecture Notes in Computer Science*, pp. 40–55. Springer Berlin / Heidelberg, 2009.

[89] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pp. 547–548, New York, NY, USA, 2007. ACM.

[90] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In Gregor Engels, Bill Opdyke, Douglas Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, Vol. 4735 of *Lecture Notes in Computer Science*, pp. 436–450. Springer Berlin / Heidelberg, 2007.

[91] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, Vol. 11, No. 2, pp. 256–290, April 2002.

[92] D. Jackson. Alloy analyzer.

[93] Martin Gogolla, Jorn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software and Systems Modeling*, Vol. 4, pp. 386–398, 2005.

[94] Anna Queralt and Ernest Teniente. Reasoning on uml class diagrams with ocl constraints. In David Embley, Antoni Olive, and Sudha Ram, editors, *Conceptual Modeling - ER 2006*, Vol. 4215 of *Lecture Notes in Computer Science*, pp. 497–512. Springer Berlin / Heidelberg, 2006.

[95] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. Ocl-lite: Finite reasoning on uml/ocl conceptual schemas. *Data & Knowledge Engineering*, Vol. 73, No. 0, pp. 1–22, 2012.

[96] H. Storrle. Semantics of control-flow in uml 2.0 activities. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 235 –242, sept. 2004.

[97] Yann Thierry-Mieg and Lom-Messan Hillah. Uml behavioral consistency checking using instantiable petri nets. *Innovations in Systems and Software Engineering*, Vol. 4, pp. 293–300, 2008.

[98] Gregor Engels, Christian Soltenborn, and Heike Wehrheim. Analysis of uml activities using dynamic meta modeling. In MarcelloM. Bonsangue and EinarBroch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, Vol.

4468 of *Lecture Notes in Computer Science*, pp. 76–90. Springer Berlin Heidelberg, 2007.

[99] Edmund M. Clarke Jr., Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 2000.

[100] Rik Eshuis. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, Vol. 15, No. 1, pp. 1–38, January 2006.

[101] Rik Eshuis and Roel Wieringa. Verification support for workflow design with uml activity graphs. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 166–176, New York, NY, USA, 2002. ACM.

[102] N. Guelfi and A. Mammar. A formal semantics of timed activity diagrams and its promela translation. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, 2005.

[103] Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne. Formal verification of tokeneer behaviours modelled in fuml using csp. In Jin Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, Vol. 6447 of *Lecture Notes in Computer Science*, pp. 371–387. Springer Berlin / Heidelberg, 2010.

[104] C.A.R.Hoare. Communicating sequential processes, 2004. `http://www.usingcsp.com/cspbook.pdf`.

[105] F.A. Kraemer, H. Samset, and R. Braek. An automated method for web service orchestration based on reusable building blocks. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pp. 262–270, 2009.

[106] F.A. Kraemer and P. Herrmann. Formalizing collaboration-oriented service specifications using temporal logic. In *Networking and Electronic Commerce Research Conference*, pp. 194–220, USA, October 2007. ATSMA Inc.

[107] Elena Planas, Jordi Cabot, and Cristina Gomez. Verifying action semantics specifications in uml behavioral models. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *Advanced Information Systems Engineering*, Vol. 5565 of *Lecture Notes in Computer Science*, pp. 125–140. Springer Berlin / Heidelberg, 2009.

[108] Elena Planas, Jordi Cabot, and Cristina Gomez. Lightweight verification of executable models. In Manfred Jeusfeld, Lois Delcambre, and Tok-Wang Ling, editors, *Conceptual Modeling – ER 2011*, Vol. 6998 of *Lecture Notes in Computer Science*, pp. 467–475. Springer Berlin / Heidelberg, 2011.

[109] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pp. 238–252, New York, NY, USA, 1977. ACM.

[110] 堀内謙二. 論理プログラムの抽象解釈を用いた解析. コンピュータソフトウェア, Vol. 11, No. 1, pp. 3–23, 1994.

[111] A. Baruzzo and M. Comini. Static verification of uml model consistency. In D. Hearnden, N. S, J. Rapin, and B. Baudry, editors, *3rd Workshop on Model Design and Validation*, pp. 111–126, 2006.

[112] Jordi Cabot and Ernest Teniente. Incremental integrity checking of uml/ocl conceptual schemas. *Journal of Systems and Software*, Vol. 82, No. 9, pp. 1459–1478, 2009.

[113] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pp. 194–206, New York, NY, USA, 1973. ACM.

[114] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pp. 1–12, New York, NY, USA, 1995. ACM.

[115] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pp. 91–103, New York, NY, USA, 1999. ACM.

[116] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, Vol. 30, No. 1, November 2007.

[117] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Cameron Foulger. Data flow and validation in workflow modelling. In *Proceedings of the 15th Australasian database conference - Volume 27*, ADC '04, pp. 207–214, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[118] Hema S. Meda, Anup Kumar Sen, and Amitava Bagchi. On detecting data flow errors in workflows. *J. Data and Information Quality*, Vol. 2, No. 1, pp. 4:1–4:31, July 2010.

[119] Harald Storrle. 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.

[120] Tabinda Waheed, Muhammad Iqbal, and Zafar Malik. Data flow analysis of uml action semantics for executable models. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, Vol. 5095 of *Lecture Notes in Computer Science*, pp. 79–93. Springer Berlin / Heidelberg, 2008.

[121] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. In Peri L. Tarr and Alexander L. Wolf, editors, *Engineering of Software*, pp. 115–141. Springer Berlin Heidelberg, 2011.

[122] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Eng. Methodol.*, Vol. 1, No. 1, pp. 21–52, January 1992.

[123] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pp. 38–48, New York, NY, USA, 1998. ACM.

[124] Bernhard Steffen. Data flow analysis as model checking. In Takayasu Ito and AlbertR. Meyer, editors, *Theoretical Aspects of Computer Software*, Vol. 526 of *Lecture Notes in Computer Science*, pp. 346–364. Springer Berlin Heidelberg, 1991.

[125] Bernhard Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, Vol. 21, No. 2, pp. 115–139, 1993.

[126] MDT project. MDT/UML2. `http://www.eclipse.org/modeling/mdt/?project=uml2`.

[127] MDT project. MDT/OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[128] AT&T Research. Graphviz. `http://graphviz.org/`.

[129] L. Frias, A. Queralt, and A. Olive. Eu-rent car rentals specification. Technical Report LSI-03-59-R, Universitat Politecnica de Catalunya, 2003.

[130] A. Queralt and E. Teniente. A platform independent model for the electronic marketplace domain. Technical report, Universitat Politecnica de Catalunya, 2005.

[131] J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Science of Computer Programming*, Vol. 68, No. 3, pp. 179–195, 2007.

[132] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of uml/ocl models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pp. 185–194, New York, NY, USA, 2010. ACM.

[133] TOPCASED project. Topcased. `http://www.topcased.org/`.

[134] Francisco Curbera, Rania Khalaf, Frank Leymann, and Sanjiva Weerawarana. Exception handling in the bpel4ws language. In WilM.P. Aalst and Mathias Weske, editors, *Business Process Management*, Vol. 2678 of *Lecture Notes in Computer Science*, pp. 276–290. Springer Berlin Heidelberg, 2003.

[135] Oliver Kopp, Rania Khalaf, and Frank Leymann. Deriving explicit data links in ws-bpel processes. In *Proceedings of the International Conference on Services Computing, Industry Track, SCC 2008*, pp. 367–376. IEEE Computer Society, 2008.

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

[137] I. Ait-Sadoune and Y. Ait-Ameur. A proof based approach for modelling and verifyingweb services compositions. In *14th IEEE International Conference on Engineering of Complex Computer Systems,*, pp. 1–10, 2009.

付録 A

抽象領域における解釈

本研究で提案した OCL 記述の解析手法（3 章）が扱う OCL 表現ノードのバリエーションと，抽象領域における解釈規則について説明する．

A.1　OCL 表現ノードのバリエーション

OCL 表現の抽象構文メタモデルを図 A.1 に示す．OCL 標準ライブラリの基本サブセットである BasicOCL パッケージは，図 A.1 の OCL 表現の 12 種類の具象クラスのうち，点線部以外の 9 種類を扱う[*1]．本手法では，この 9 種類の OCL 表現について，OCL 表現ノードの解釈の規則を定義する．

OCL 表現ノードは，その種類とノード名の組み合わせ（バリエーション）によって

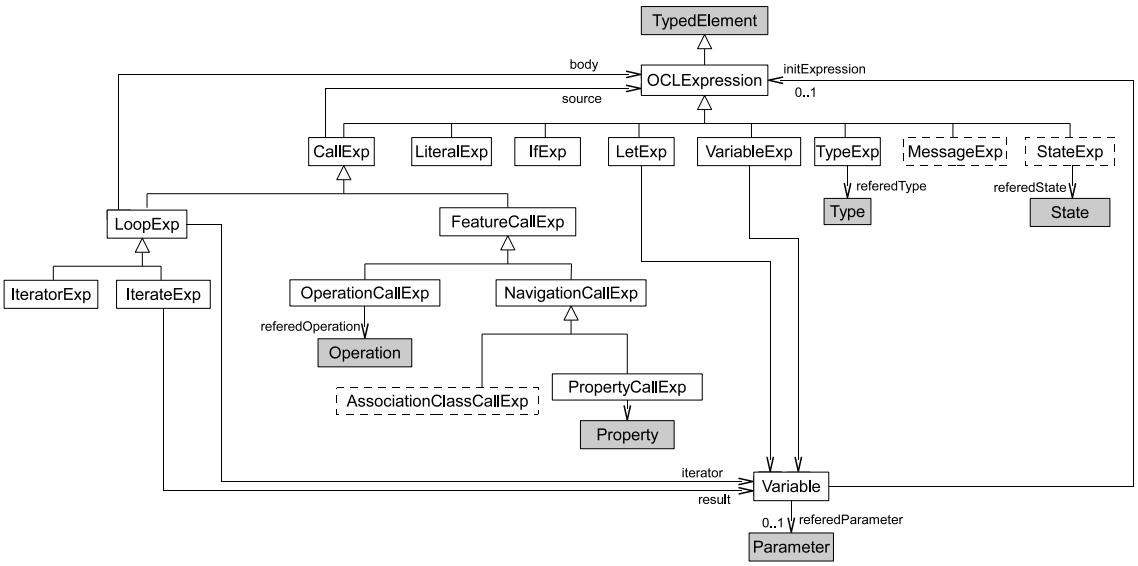


図 A.1. OCL 表現の抽象構文メタモデル

[*1] 点線部の *AssociationClassCallExp*，*StateExp*，*MessageExp* は，それぞれ関連クラス，状態，メッセージを扱う OCL 表現である．

表 A.1. OCL 表現ノードのバリエーション

OCL 表現	ノード名	解釈規則	OCL 表現	ノード名	解釈規則	OCL 表現	ノード名	解釈規則
LiteralExp	リテラル	x		=	x	*OperationCallExp*	size	x(*Set*)
IterateExp	iterate	-		<>	x	(*Collection*)	includes	x(*Set*)
IteratorExp	exists	x(*Set*)	*OperationCallExp*	+	x		excludes	x(*Set*)
	forAll	x(*Set*)	(*PrimitiveType*)	-	x		count	x(*Set*)
	isUnique	-		*	x		includesAll	x(*Set*)
	any	-		-(negative)	x		excludesAll	x(*Set*)
	one	x(*Set*)		/	x		isEmpty	x(*Set*)
	collect	x(*Set*)		abs	x		notEmpty	x(*Set*)
	select	x(*Set*)		floor	x		sum	-
	reject	x(*Set*)		round	x		product	-
	collectNested	-		max	x		union	x(*Set*)
	sortedBy	-		min	x		intersection	x(*Set*)
IfExp		x		<	x		ñ	x(*Set*)
LetExp		x		>	x		including	x(*Set*)
VariableExp	変数名	x		>=	x		excluding	x(*Set*)
TypeExp	型名			<=	x		symmetricDifference	x(*Set*)
PropertyCallExp		x		div	x		flatten	-
OperationCallExp	oclIsNew	x		mod	x		asSet	x(*Bag*)
	oclIsUndefined	-		concat	x		asOrderedSet	-
	oclIsInvalid	-		size	x		asSequence	-
	allInstances			substring	x		asBag	-
	oclAsType	-		toInteger	x	*OperationCallExp*	subSequence	-
	oclIsTypeOf	-		toReal	x	(*Sequence*)	append	-
	oclIsKindOf	-		ToUpper	x		prepend	-
	oclInState	-		toLower	x		insertAt	-
	hasReturned	-		and	x		subOrderedSet	-
	result	-		or	x		at	-
	isSignalSent	-		xor	x		indexOf	-
	isOperationCall	-		not	x		first	-
				implies	x		last	-

意味が異なるため，バリエーション毎に解釈規則を定める必要がある．ノード名は，*IteratorExp*/*IterateExp*/*OperationCallExp* ノードでは演算の名前を，*LiteralExp* ノードでは参照するリテラルを，*TypeExp* ノードでは参照する *Type* の名前を，*VariableExp* では参照する変数の名前を，それぞれ表す．*IfExp*/*LetExp*/*PropertyCallExp* ノードは無名である．BasicOCL における 89 のバリエーションを表 A.1 に示す．手法は 57 のバリエーションの解釈規則を定義し，その中の 21 は適用対象の型を限定する．解釈規則を定義する場合は “x”，解釈規則を定義しない場合は空欄，手法の対象外である場合は “-”（ハイフン）で表す．”x(T)” は，型 T についてのみ解釈規則を定義することを表す．

表 A.1 中の “x” 以外のバリエーションについて，手法で解釈規則を定義しない，あるいは対象外とする理由を以下に説明する．

- *TypeExp* ノードは，UML モデルで静的に与えられた *Type* の参照を表し，事後条件を満たすオブジェクト状態の有無に直接影響を与えないため，値の割り当てが不要である．
- 演算 `allInstances` の *OperationCallExp* ノードは，子ノードが常に *TypeExp* ノードであり，子ノードの値の解釈が不要である．

- 演算 `oclIsUndefined` の *OperationCallExp* ノードは未定義値の評価を表す．該ノードが複数の親ノードを持つ合流点ノードである場合には，全ての親ノードからの経路上で未定義値 U を割り当てるケースにおいてのみ，評価結果が真となる．しかし，3.2.3 項で述べた静的解析手法では，一部の経路で T/F の値を持つケースを許容しており，正しい計算結果が得られない可能性があるため，手法の対象外とする．このバリエーションを扱うためには，手法の拡張が必要であり，今後の課題である．
- 演算 `oclIsInvalid` の *OperationCallExp* ノードは例外の評価を表すが，本手法では事後条件で例外を扱わないため，このバリエーションは不要である．
- 演算 `oclAsType`，`oclIsTypeOf`，`oclIsKindOf` の *OperationCallExp* ノードは，型のキャストや適合を評価する表現であるが，本手法が扱うコンポーネントモデルは継承関係を含まないため，これらのバリエーションは不要である．
- 演算 `oclInState`，`hasReturned`，`result`，`isSignalSent`，`isOperationCall` の *OperationCallExp* ノードは，BasicOCL の対象外である状態とメッセージに関する演算を表すため，これらのバリエーションを扱わない．
- *IteratorExp*/*OperationCallExp* ノードは，*Collection* 全般に対して定義されるが，手法では頻度の高い *Set* の演算に限定し，*Bag* の演算は扱わない．ただし，演算 collect は適用結果が *Bag* であるが，実モデルで頻用されている．また *Bag* の演算 asSet は，collect の演算結果を *Set* に変換するために併用される．そこで手法は，特例としてこれらの演算の解釈規則を定義する．
- 演算 `sortedBy`，`any` の *OperationCallExp* ノードは，*Sequence* の演算を用いて定義されるため，扱わない．

IterateExp ノードと一部の *IteratorExp*/*OperationCallExp* ノード（`isUnique`, `collectNested`, `sum`，`product`）を扱わない理由は，次の解釈規則の説明の中で述べる．

A.2　解釈規則

OCL 仕様書（OMG [27]）の 10 章に規定されているように，OCL 表現ノードの値は，オブジェクト状態，すなわち環境に対して動的に評価を行うことにより決定される．一方，本手法では，各ノードが表す意味に基づき，親子ノードが取り得る抽象値の組み合わせを解釈規則として静的に定める．以下に，提案手法が定める解釈規則を説明する．57 のバリエーションの全ての解釈規則を網羅的に列挙して説明するかわりに，OCL 言語仕様から解釈規則を導くための手順を定義して，幾つかのバリエーションの解釈規則を例示する．

OCL 表現ノードの解釈規則を求める手順を以下に示す．

1. OCL 仕様書（OMG [27]）の 11 章と Annex A で定められたノードの意味論に従って，ノードと，その子ノードが取り得る抽象値の組み合わせを求める．
2. 手順 1 で求めた，親子ノードの抽象値の組合わせから，親ノードの値が U のケースを除外する．

表 A.2. 各種 OCL 表現ノードの解釈

	LiteralExp	*OperationCallExp* (oclIsNew)	*PropertyCallExp*	*LetExp*	*IfExp*			*VariableExp*	*OperationCallExp* (+)	
T	T	T	T	T	T/F	T/U	U/T	T	T/T/F	F/T/T
F	F	T	T	F	T/F	F/U	U/F	F	T/F	T/F

3. 親ノードの値（T/F）に対して子ノードが取り得る値の組合わせを解釈規則として求め，表 3.4 の形式にまとめる．

手順 1 は対象ノードのバリエーションと型に依存し，手順 2，3 は汎用の処理である．以下では，手順 1 についてバリエーション毎に説明する．

LiteralExp ノードでは，リテラルの値を固定的な具体値として，抽象化写像 α を用いてノードの抽象値を求める．演算 `oclIsNew` の *OperationCallExp* ノードでは，子ノード source が表すオブジェクトが，操作実行中に生成されたか否かによって，親ノードが抽象値 T，F の両方を取りうる．また以下のノード群では，親ノードが子ノードや関連先のクラスの値を引き継ぐ．その引き継ぎ方に従って，抽象値の組み合わせを求める．

- *IfExp* ノードは，子ノード condition の Boolean 値によって，thenExpression/elseExpression の何れか片方の子ノードの値を引き継ぐ．
- *LetExp* ノードは子ノード in の値を引き継ぐ．
- *PropertyCallExp* ノードは，子ノード source が表すオブジェクトの *Property* の値を引き継ぐ．
- *VariableExp* ノードは，参照する *Variable* の値を引き継ぐ．*Variable* の値は，関連先の initExpression ノードや referedParameter の値に対応する．

これらのバリエーションの解釈規則を表 A.2 に記載する．なお *LiteralExp, PropertyCallExp, VariableExp* については，子ノードの代わりに，リテラル，*Property*, *Variable* の抽象値を，交点のセルに示している．

PrimitiveType 型の *OperationCallExp* ノードでは，3.2.1 項で定義した抽象化写像 α に基づいて，具体領域における子ノードの領域分割を行い，子ノードの領域の組み合わせ毎に親ノードの取り得る値を求める．例えば *Integer* 型の演算 + の *OperationCallExp* ノードの場合は，具象領域を $P = (0, +\infty)$，$N = (-\infty, 0)$，$Z = [0, 0]$，*OclVoid* の 4 つの領域に分割する．演算 + の意味から，或る $p, p' \in P$，$n, n' \in N$，$z, z' \in Z$，$v \in OclVoid$ に対して，$p + z \in P$，$p + p' \in P$，$p + n \in P(p > -n$ の場合)，$p + n \in N(p < -n$ の場合)，$p+n \in Z(p = -n$ の場合)，$n+z \in N$，$n+p \in P(p > -n$ の場合)，$n+p \in N(p < -n$ の場合)，$n + p \in Z(p = -n$ の場合)，$n + n' \in N$，$z + p \in P$，$z + n \in N$，$z + z' \in Z$，$v+? \in V$，$? + v \in V$ の演算結果が考えられる．ここで記号 ? は p, n, z, v の何れかを表す．以上の具体値の組み合わせを抽象化して，更に手順 2，3 を実施すると，表 A.2 の解釈規則が得られる．

次に，*Set* に対する *IteratorExp*/*OperationCallExp* ノードの手順 1 を説明する．そのために先ず *IterateExp* について説明する．OCL 表現 $e_1 \in Set(\mathrm{T})$ の `iterate` 演算の意味は，次の式で表される[*2]．

$$I[[e_1\text{->iterate}(v_1; v_2 = e_2 | e_3)]] = \begin{cases} I[[e_2]] \\ \qquad if\ I[[e_1\{v_2/e_2\}]] = \phi, \\ I[[Set\{x_2, .., x_n\}\text{->iterate}(v_1; v_2 = e_3\{v_1/x_1, v_2/e_2\} | e_3)]] \\ \qquad if\ I[[e_1\{v_2/e_2\}]] = \{x_1, .., x_n\}. \end{cases}$$

`iterate` 演算は，e_1 の各要素 v_1 について e_3 の評価を繰り返し，繰り返しの度に評価結果を変数 v_2 に保持することを表す．v_2 の初期値は e_2 であり，繰り返し完了時の v_2 の値が演算の結果を表す．一般的に子ノード e_3 の値は，e_1 の要素について評価を繰り返す度に更新されるため，取り得る値を静的に一意に定めることが出来ない．従って *IterateExp* ノードは，本手法で扱うことが出来ない．

Set に対する全ての OCL 表現ノードの演算は，`iterate` 演算を利用して定義されており，一般的には親子ノードの値を静的に定めることは出来ない．ただし中には，`iterate` 演算を利用する際に引数として与える OCL 表現が，次の 2 つの条件を満たすような演算が存在する．

条件 1：或る繰り返しにおいて，v_2 の値が変化するならば，その時の e_3 が一定値 c_3 を取り，以降の繰り返しを通じて v_2 は一定値 c_3 を取る．

条件 2：演算を通じて v_2 の値が不変ならば，e_3 が常に一定値 c_3' を取る．

そのような演算では，`iterate` 演算ノード，e_1，e_3 の値の組として，条件 1 に対応して (c_3, S, c_3) を，条件 2 に対応して (e_2, S, c_3') を，さらに e_1 が空の場合に対応して $(e_2, Set\{\}, OclVoidValue)$ を，静的に定めることが出来る．ただし S は条件 1，2 を満たす非空集合である．例えば `exists` 演算：

$$e_1\text{->exists}(v_1 | body) = e_1\text{->iterate}(v_1; v_2 : Boolean = \mathrm{false} \mid v_2 \text{ or } body)$$

の場合は $c_3 =$ `true`，$c_3' =$ `false` である．これらの値から，`exists` 演算ノード，e_1，*body* の値の組を (`true`, S, `true`)，(`false`, S, `false`)，(`false`, $Set\{\}, OclVoidValue$) と定義する．さらに，これらに対応する抽象値の組み合わせを求めて，手順 2，3 を実施することによって，解釈規則が得られる．同様に演算子 `forAll` の解釈規則を求めることが出来る．演算子 `size`，`count` では，具体領域では v_2 の変化の度に値が 1 ずつ増加するため，条件 1 を満たさないが，抽象領域では常に T となり条件 1 を満たす．そこで，抽象領域上での値の組み合わせを直接求めて，解釈規則を定める．これらの演算ノードの解釈規則を表 A.3 に示す．

演算 `collect` は，演算 `collectNested`/`flatten` を用いて定義される[*3]．

[*2] $e\{v'/e'\}$ は，OCL 表現 e に出現する変数 v' を e' で置き換えて得られる OCL 表現を表し，$I[[e]]$ は，ある環境における e の値を表す．

[*3] `flatten` 演算は，演算を適用する $src \in Bag$ の要素が *Collection* 型の場合には，src の全ての要素の全ての要素を含む *Bag* を返し，src の要素が *PrimitiveType* 型の場合には src を返す．

表 A.3. *Collection* 型の演算ノードの解釈

	IteratorExp (exists)		*IteratorExp* (forAll)		*OperationCallExp* (size,asSet)	*IteratorExp* (count)		*IteratorExp* (collect:*PrimitiveType*)		*IteratorExp* (collect:*Collection*)	
T	T	T	T/F	T/U	T	T/T	T/F	T	U	T	T
F	T/F	F/U	T	F	F	F/T/T	U/T/F	F	U	T/F	F/U

$$e_1\text{->collect}(v_1|body) = e_1\text{->collectNested}(v_1|body)\text{->flatten}(),$$
$$e_1\text{->collectNested}(v_1|body) =$$
$$e_1\text{->iterate}(v_1; v_2 : Bag(body.type) = Bag\{\}|v_2\text{->including}(body)).$$

演算 `collectNested` では，演算 `iterate` の繰り返しの度に *body* ノードの値が *Bag* である v_2 に追加されるので，条件 1 を満たさない．しかし抽象領域上では，*body* が *PrimitiveType* 型の場合は，$e_1 \neq \phi$ ならば演算 `collect` の結果は *body* の値に依らず常に T である．そこで `collect` 演算ノード，e_1，*body* の値の組として，条件 1 に対応して (T,T,U) を，e_1 が空の場合に対応して (F,F,U) を定義する．また *body* が *Collection* 型の場合は，$body \neq \phi$ となる繰り返しが存在すれば，演算 `collect` の結果は T となり，*body* が常に ϕ ならば，演算結果は F である．従って，`collect` 演算ノード，e_1，*body* の値の組として，条件 1 に対応して (T,T,T) を，条件 2 に対応して (F,T,F) を，e_1 が空の場合に対応して (F,F,U) を定める．演算 `collect` の解釈規則を表 A.3 に示す．

演算 `asSet` は，演算を適用する $src \in Bag$ が非空の場合には，*src* の要素のみを含む集合を返し，*src* が空の場合には空集合を返す．従って表 A.3 に示す解釈規則が得られる．

演算 `isUnique`/`collectNested`/`sum`/`product` は，演算 `iterate` を用いて定義されているが，前述の条件 1，2 を満たさないため，本手法では扱うことが出来ない．

その他の *IteratorExp*/*OperationCallExp* の演算は，既に説明した演算を利用して定義されている．例えば *src*->isEmpty() の演算結果は *src*->size() = 0 の値と等しい．そのような演算ノードの解釈規則は，既出の解釈規則を用いて構成することが出来る．

付録 B

OCL 記述の解析の計算量

本研究で提案した OCL 記述の解析手法（3 章）の計算量の見積り方法を以下に説明する．5.1.5 節の表 5.1 で定義した抽象構文木の属性を用いて，計算量を表現する．

抽象解釈（Step 1.1）

3.2.3 項のサブルーチン OBTAIN-POSSIBLE-VALUES は，抽象構文木の下層ブロック間を，末端ノードから上向きに一度，根ノードから下向きに一度，走査を行い，下層ブロックの部分集合を返す．従って，その時間計算量は $\mathrm{O}(N_b + L_b)$，空間計算量は $\mathrm{O}(N_b)$ である．そしてアルゴリズム CALCULATE-ABSTRACT-VALUES は，事後条件毎に 1 回実行され，分岐点と合流点の下層ブロックの N_c 個の一意な組み合わせ毎に，OBTAIN-POSSIBLE-VALUES を呼び出し，得られる結果の集合を返す．故に時間計算量は $\mathrm{O}((N_b + L_b)N_c)$，空間計算量は $\mathrm{O}(N_b \times N_c)$ である．

起源の解析（Step 1.2）

この解析に用いるアルゴリズム OBTAIN-ORIGINS（図 3.7）は，ノード間のリンクを辿って各ノードを最大 1 回訪問するので，時間計算量は $\mathrm{O}(N + L)$ である．また各ノードの起源には構造モデル上の複数の名前付き要素が含まれ得るが，現実のモデルではその要素数は高々数個なので， 起源を保持するために必要な記憶容量は $\mathrm{O}(1)$ である．故に空間計算量は $\mathrm{O}(N)$ である．

データ流向の特定（Step 1.3）

Step 1.2 で抽出した $\mathrm{O}(1)$ 個の代入ノード，取得ノード，照合ノードの候補について，データ流向を特定し，表 3.5 の決定表を用いて，代入ノード，取得ノード，照合ノードを決定する．データ流向の特定とノードの決定は $\mathrm{O}(1)$ 回の演算で実行できるため，必要な計算ステップ数と記憶容量は共に $\mathrm{O}(1)$ である．

確定性の判定（Step 2.1）

この解析に用いるアルゴリズム DECIDE-CERTAINTY は，$\mathrm{O}(1)$ 個の出力ノード候補毎に実行され，Step 1.1 の解析で得られる，最大 N_c 個の有効な下層ブロック集合の各々について，3.2.3 項のサブルーチン OBTAIN-POSSIBLE-VALUES を 1 回実行する．故に時間計算量と空間

計算量は Step 1.1 の計算量と等しい.

値の対応付け（Step 2.2）

この解析に用いるアルゴリズム OBTAIN-VALUE-MAPPING は，O(1) 個の照合出力ノード候補と照合ノードの組み合わせ毎に実行され，Step 1.1 の解析で求めた最大 N_c 個の有効な下層ブロック集合の各々について，3.2.3 項のサブルーチン OBTAIN-POSSIBLE-VALUES を 4 回実行する．故に時間計算量と空間計算量は Step 1.1 の計算量と等しい.

含意判定（Step 2.3）

Step 2.2 で抽出した O(1) 個の出力ノード候補と照合ノードの組み合わせの値の対応 $f_{\mathrm{E_0},\mathrm{E_c}}$ を，表 3.6 の成立パターンに対して，O(1) 回の演算でマッチングする．故に時間計算量と空間計算量は共に O(1) である.

図目次

表目次