コード検査ツールの開発における
検査のためのプログラムの自動生成に関する研究

M2012MM021 水谷誠孝
指導教員：野呂昌満

1　はじめに

コード検査ツール（以下，CDI ツール）は，ソースコードを対象として静的に検査を行うことでソフトウェアの潜在的な欠陥を検出するツールである．本研究室では，Java を対象とする CDI ツール (以下，JCI) の開発を行ってきた [4][5]．JCI の検査仕様は自然言語で記述されてきた．検査処理のためのコンポーネント (以下，検査処理コンポーネント) が過去の JCI の開発実績を通じて定義されている．

CDI ツールの検査に対する要求はツールの利用者によって変化する．多くの CDI ツールは利用者による検査のカスタマイズ機能を備えておらず，その機能は限定的である．検査機能をカスタマイズするためには CDI ツールのデータ構造に習熟している必要があるので，ツールの利用者による検査機能のカスタマイズは困難である．ツールの利用者による検査機能のカスタマイズを実現するためには，仕様記述から検査のためのプログラム（以下，検査コード）を生成する枠組みが必要である．Noro らの論文 [2] では，カスタマイズ可能な CDI ツールの開発を目指し，JCI の新たなアーキテクチャの構築と状態遷移モデルを用いた検査仕様記述の提案が行われた．

本研究の目的は，検査仕様記述から自動的に検査コードを生成する枠組みの設計，実現である．検査機能のカスタマイズの枠組みを実現することによって，検査に対する要求の変化に対して柔軟な CDI ツールを実現する．

本研究では，状態遷移モデルを用いた検査仕様記述から検査コードを自動生成する．Hallem らの論文 [3] で採用された検査仕様記述を基に，検査仕様を状態遷移モデルを用いて記述する．ミーリ型の状態遷移機械として，抽象構文木やフローグラフといった，内部形を順方向に辿って出現した構文要素をイベントとして受理する．受理したイベントに対応したアクションを起動し，アクションに記述された処理コードを基に，検査処理を行う．検査処理では構文要素の情報を利用するので，検査処理を記述するためには内部形への理解が必要である．本研究では，検査処理を簡易化することで検査処理コンポーネントの標準化を行った．標準化した検査処理コンポーネントを利用する事によって内部形への理解は不要になったので，検査処理の記述が容易になった．

本研究の成果は，状態遷移モデルを用いた検査仕様記述から検査コードを自動生成するための枠組みを設計したことである．アーキテクチャと検査仕様記述との対応関係を整理し，出力される検査コードの定型であるプラットフォームコードの設計を行った．状態遷移モデルのアクション部分の記述には VDM++を利用し，検査コードの自動生成に関する考察を行った．

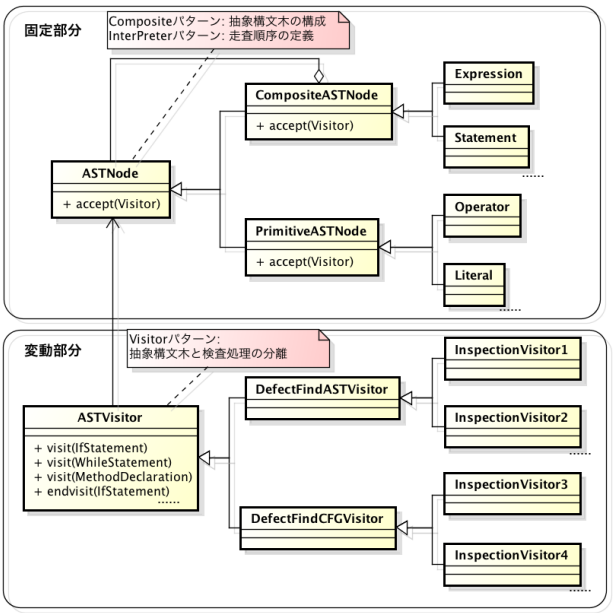

図 1　JCI のアーキテクチャ

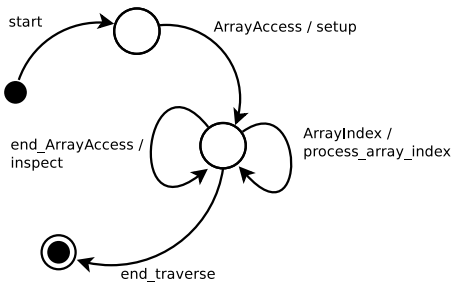

図 2　検査仕様記述の例

2　JCI の概要

JCI は，Java ソースコードから抽象構文木，フローグラフを構築し，抽象構文木，フローグラフを走査することで欠陥を検出するツールである．現在，JCI では 35 個の検査項目が実現されている．JCI のアーキテクチャを図 1 に示す．JCI のアーキテクチャには GoF デザインパターンが適用されている．特に，Visitor パターンを適用する事でデータ構造と検査処理を分離し，開発者による検査処理の変更に柔軟である．

2.1　検査仕様記述

Noro らの論文で提案された検査仕様記述は，状態遷移モデルを用いて可視化されている．状態遷移モデルを用いる事で，理解しやすく，記述しやすい検査仕様になっている．検査仕様記述の例を図 2 に示す．

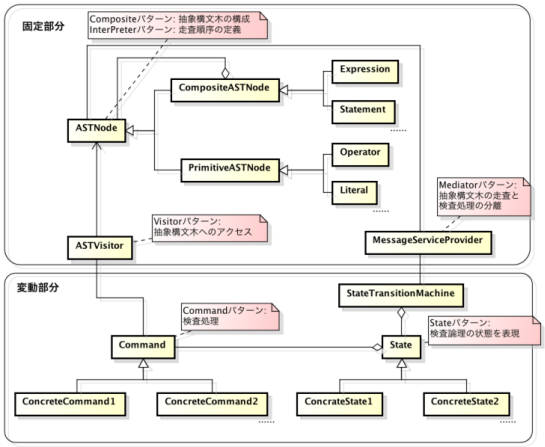

図 3 JCI の新たなアーキテクチャ

2.2 新たなアーキテクチャ

Noro らの論文で構築された JCI の新たなアーキテクチャには，アスペクト指向技術が適用されている．GoF デザインパターンを適用して設計されたアーキテクチャを図 3 に示す．

新たなアーキテクチャは状態遷移モデルを用いた検査処理を実現する．新たなアーキテクチャに適用されている GoF デザインパターンを以下に示す．

- Composite パターン
- Interpreter パターン
- Visitor パターン
- Command パターン
- Mediator パターン

抽象構文木の構造は Composite パターンを用いて実現されている．構文木とフローグラフに対する走査と走査したノードの通知処理は Interpreter パターンを用いて実現されている．状態遷移モデルを表現するために State パターンと Command パターンが利用されている．State パターンは状態による振る舞いを，Command パターンは検査処理のアルゴリズムを実現している．Visitor パターンは構文木のデータ構造へのアクセスに利用される．Mediator パターンを適用し，構文木とグラフの走査と状態遷移モデルの遷移を仲介することで，走査処理と検査処理を分離している．走査処理と検査処理の分離をしており，データ構造と検査処理を分離しているので，検査処理の変更や追加に柔軟である．

3 状態遷移モデルを用いた検査仕様記述

Hallem らの論文で採用された検査仕様記述を基に，状態遷移モデルを用いて検査仕様を記述した．Noro らの論文で提案された検査仕様記述では，検査対象となるノードが繰り返し出現した際に正しく検査を行えないことが分かった．本研究では，検査に関わるイベントと検査の関わらないイベントを区別して記述した．検査仕様を記述した結果，検査内容の種類ごとに，新たな状態遷移モデルのパターンで検査仕様を記述できる事が分かった．状態遷移モデルのパターンを図 4，図 5 に示す．

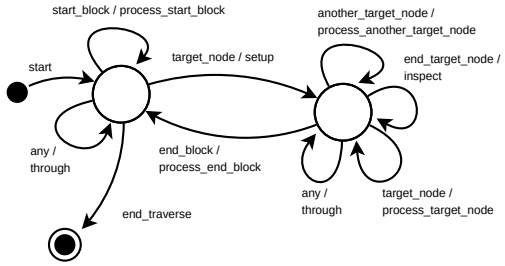

図 4 抽象構文木の構造に関する検査の仕様記述のパターン

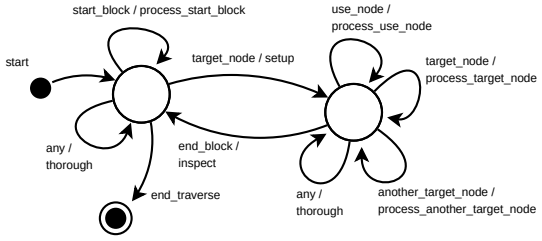

図 5 データフローに関する検査の仕様記述のパターン

target_node，use_node および another_target_node は着目する構文要素の出現を表し，end_target_node は着目する構文要素の終了を表す．start_block は検査する範囲の開始，end_block は検査する範囲の終了を表す．any は着目しない構文要素を表し，end_tarverse は検査の終了を表す．各イベントに対応するアクションでは，ネストのカウントなどの検査処理を行う．

4 検査処理コンポーネントの標準化

検査処理を簡易化することで検査処理コンポーネントの標準化を行った．検査処理では内部形の情報を利用するので，ツールの利用者が理解して記述するのは困難である．JCI の既存の検査処理と検査処理コンポーネントを整理した結果，検査処理コンポーネントは複数の異なる構文要素から利用されるという特徴があることがわかった．標準化する際は，ある検査処理についてその検査処理が利用される構文要素に着目した．また，検査処理の処理内容には以下の特徴がある事が判った．

- 構文要素の評価
- 構文要素の情報の取得
- 構文要素の情報の判断

これらの特徴を基に，内部形への理解が不要になるように，検査処理コンポーネントの標準化を行った．標準化してまとめた結果の一部を表 1 に示す．標準化によって検査処理の記述を簡易化し，検査処理を記述する際に内部形の理解が不要になった．

5 アクション部分の記述

本章では，検査仕様記述のアクション部分について議論する．アクション部分への要求と考えられる記述方法を整理し，それらの関連を考察することによってアクション部分の記述方法を決定した．

表 1 標準化した検査処理コンポーネント

コンポーネント名	処理内容	着目する構文要素
evaluateCondition	真偽値を取得する	ループ文，分岐文
evaluateValue	計算結果を取得する	代入式，変数
getArraySize	配列長を取得する	配列アクセス式
getCondition	条件式を取得する	ループ文，if 文，switch 文
getExpression	式を取得する	前置式，後置式
getLeftHand	左側の式を取得する	中置式，代入文
getLoopCount	ループ回数を取得する	ループ文
getMethodName	メソッド名を取得する	使用点，
getModifier	修飾子を取得する	変数宣言文
getName	名前を取得する	変数宣言文，ラベル参照，使用点
getOperator	演算子を取得する	前置式，中置式，後置式
getReachDef	到達定義を取得する	変数宣言，使用点
getRightHand	右側の式を取得する	中置式，代入文
getType	型を取得する	変数宣言，使用点
getVariable	変数を取得する	式
isSameType	同じ型かどうか判断する	変数，式

5.1 アクション部分の記述に求められる事

アクション部分で行なわれる検査処理の特徴と CDI ツールのカスタマイズの要求から，アクション部分の記述に求められる事を整理する．アクション部分は標準化した検査処理コンポーネントの組み合わせによって記述する．CDI ツールに対するカスタマイズの要求の点から整理した，アクション部分に求められる事は，記述の厳密さ，記述しやすさ，理解しやすさである．曖昧な仕様は誤った実現を引き起こす可能性がある．誤った実現ではカスタマイズの要求を満たす事はできないので，本研究では，特に記述の厳密さを重視する．

5.2 考えられる記述

アクション部分の記述に利用できると考えられる記述について整理する．考えられる記述方法を以下に示す．

- 形式言語
  - VDM++
  - CSP 記述
- 図式言語
  - シーケンス図
  - アクティビティ図

仕様記述に用いられる方法のうち，形式言語と図式言語が利用できると考えられる．また，それらの中から，コンポーネントの組み合わせで記述するという点から，VDM++，CSP 記述，シーケンス図，アクティビティ図が利用できると考える．形式言語を用いて記述することで，アクション部分を厳密に記述する事が出来ると考える．図式言語を利用することで，アクション部分の記述を容易に行なう事ができ，その記述は理解しやすいと考える．

5.3 アクション部分の記述方法

整理した要求と考えられる記述との関連を考察することで，アクション部分の記述に用いる記述方法を決定する．アクション部分の記述に求められる事のうち，記述の厳密さは形式言語である VDM++と CSP 記述が優れている．記述しやすさ，理解しやすさはシーケンス図とアクティビティ図が優れている．本研究では，記述の厳密さを重視するので，VDM++と CSP 記述が適している．VDM++と CSP 記述を比較した際，記述の厳密さ以外の要求である記述のしやすさと理解のしやすさを考慮すると，VDM++が適している．本研究では VDM++を利用してアクション部分を記述する．VDM++によるアクション部分の記述の例を以下に示す．

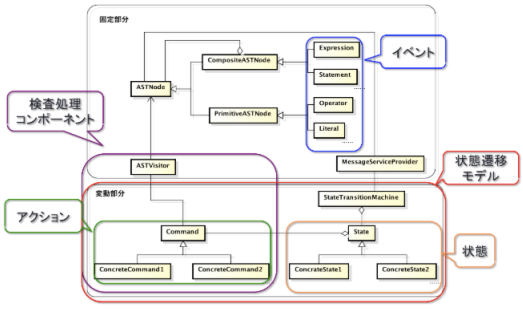

図 6 仕様記述とアーキテクチャとの対応関係

アクション部分の記述例

```
class setup
 operations
  public execute:inspect0401_1_2*Event==>()
  execute(ins,ev) == (
   ins.setArraySize(cmpnt.getArraySize(ev)); );
end setup
```

6 検査コード自動生成の枠組み

本章では，検査仕様記述から検査コードを生成するツールの枠組みの設計を行なう．検査仕様記述とアーキテクチャの対応関係から自動生成箇所を明確にし，プラットフォームコードを設計し，自動生成の枠組みの設計を行なう．

6.1 生成箇所の明確化

仕様記述とアーキテクチャの対応関係から，コード自動生成を行なう箇所を明確にする．検査仕様記述とアーキテクチャの対応関係を図 6 に示す．

状態遷移モデル全体がアーキテクチャの変動部分と対応しており，自動生成範囲である．仕様記述の中の状態の部分はアーキテクチャ中の State と対応している．仕様記述の中のアクション部分はアーキテクチャ中の Command と対応している．標準化した検査処理コンポーネントは，Visitor，Command として対応している．簡潔化した処理をまとめるために Command を用いる．検査処理コンポーネントは，複数の構文要素から利用され，再利用されるであるという点から，固定部分の Visitor としてまとめる．

6.2 検査コード自動生成の枠組みの設計

明確化した生成箇所と仕様記述を基に，検査コード自動生成の枠組みを設計する．設計した枠組みを図 7 に示す．

本研究で設計した検査コード自動生成の枠組みでは，検査仕様記述を入力として，検査コードを出力する．アクション部分の記述に対応するコード片は，VDMTools の

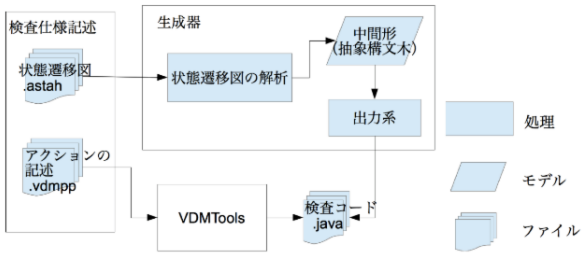

図 7 検査コード自動生成の枠組み

Java コードジェネレータを利用して生成する．本研究で設計した枠組みでは，状態遷移モデルから検査コードの自動生成を行なうツールの設計と実現が必要である．入力された状態遷移モデルを解析し，中間形である抽象構文木へと変換する．出力系で抽象構文木を走査して処理し，コード片を出力する．状態遷移モデルに対応するコード片とアクション部分の記述に対応するコード片を合わせる事で，最終的な検査コードを得る．

7 考察

7.1 検査仕様記述に関する考察

状態遷移モデルを用いた検査仕様記述に関して考察する．本研究では，検査仕様記述に状態遷移モデルを用いた．記述にあたっては，検査を行う際に着目するイベントと着目しないイベントをそれぞれ区別した．また，検査対象となるノードが繰り返し出現した際にも正しく検査を行うことができるように，検査を行う範囲と検査対象となるノードの出現と終了に着目して記述した．結果，JCI のすべての検査仕様を状態遷移モデルを用いて記述することができたので，状態遷移モデルによる検査仕様記述は妥当であると考える．

7.2 検査コードの自動生成に関する考察

検査コードの自動生成に関して考察する．本研究では，状態遷移モデルのアクション部分の記述に VDM++を用いた．VDM++から生成されるプログラムコードの例を以下に示す．

VDM++から生成されるプログラムコードの例

```
public class setup {
  private void vdm_init_setup () {}
  public setup () throws CGException {
    vdm_init_setup();
  }
  public void execute(final inspect0401_1_2 ins,
  final Event ev) throws CGException {
      ins.setArraySize(
      cmpnt.getArraySize(ev));
  }
}
```

すべての検査仕様のアクション部分について VDM++で記述できることを確認した．記述する際は，検査処理コンポーネントを用いて，内部形への理解が不要であることを確認した．結果，本研究における検査処理コンポーネントの標準化とアクション部分の記述は妥当であると考える．

7.3 検査処理コンポーネントの標準化に関する考察

検査処理コンポーネントの標準化による影響について考察する．状態遷移モデルを用いた検査仕様記述において，イベントは内部形を辿って出現した構文要素として定義した．無限ループの検出を行う検査では，for 文，while 文，do-while 文といった複数の構文要素を検査対象としている．複数の構文要素が対象となる際は，構文要素に対応してそれぞれに状態遷移モデルが必要である．アクションについても，構文要素に対応してそれぞれにアクションが必要となる．本研究で行った検査処理コンポーネントの標準化によって，アクション部分の記述において for 文など内部形の違いを意識せずに記述することが可能となった．これにより，アクション部分の記述を for 文などでまとめて記述することが可能となった．イベントについて for 文などをまとめて loop 文と定義することによって，１つの状態遷移モデルで複数の構文要素を対象とする検査項目を記述する事ができた．結果，状態遷移モデルの数が削減できた．

8 おわりに

本研究では，カスタマイズ可能な CDI ツールを実現するために，検査コードの自動生成の枠組みの設計を行なった．検査仕様記述には状態遷移モデルを用いた．検査処理コンポーネントの標準化によって，内部形に対する理解が不要になり，容易に検査仕様を記述できるようになった．今後の課題として，プラットフォームコードの設計と実現，生成器の設計と実現が挙げられる．

参考文献

[1] E. Gamma, J. Vlissides, R. Helm, and R. Johnson, *Design Pattern Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] M. Noro, and A. Sawada,“ Aspect-Oriented Software Architecture for CDI Tools: toward PLSE Construction, ” *Technical Report of the Nanzan University Academic Society Information Sciences and Engineering*, NANZAN-TR-2012-02, 2012.

[3] S. Hallem, B. Chief, Y. Xie, and D. Engler,“ A System and Language for Building System-Specific, Static Analyses, ” *PLDI'02 Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 69-82, 2002.

[4] 浦野彰彦，沢田篤史，野呂昌満，蜂巣吉成，張漢明，吉田敦,“ デザインパターンを用いたソースコードインスペクションツールのソフトウェアアーキテクチャ設計, ”第 17 回ソフトウェア工学の基礎ワークショップ FOSE2010，2010.

[5] 沢田篤史，野呂昌満，蜂巣吉成，張漢明，吉田敦，長大介，浦野彰彦,“ ソースコードインスペクションツールのためのソフトウェアアーキテクチャの設計と進化, ”日本ソフトウェア科学会編コンピュータソフトウェア，vol. 28，no. 4，pp. 241-261，2011.