コード検査ツール開発における検査仕様記述の提案と処理コードの自動生成に関する研究

M2011MM036 金賢修
指導教員：野呂昌満

1　はじめに

ソフトウェアインスペクションとは，仕様書やプログラムなどの成果物を静的に検査することで，誤りや不具合を発見する手法である．我々は，Java を対象としたソースコードインスペクションツール，Java Code Inspector(以下，JCI) の開発を行なってきた [2]．JCI は，Java のソースコードに対して静的解析を行ない，欠陥の可能性がある構文や制御フロー，データフローのパターンを発見，指摘するツールである．我々は，開発の長期化や要求の変化に対応するために，保守性や再利用性を重視し，Product Line Software Engineering に基づいたアーキテクチャ中心開発を行なってきた．JCI の保守開発では，検査項目の追加や修正が頻繁に行なわれている．開発者はその都度自然言語で書かれた検査仕様を基に手作業で記述しており，労力がかかっている．また検査仕様を基にした検査項目の開発において検査仕様と誤った検査処理コードを開発し，修正を余儀なくされ余計なコストが生じるリスクが存在する．

本研究の目的は，新たな検査項目の追加や修正を支援するために検査仕様記述手法を提案し，検査仕様記述を用いた検査処理コードの自動生成を実現することである．これにより検査作成の労力を削減し，誤った検査処理コード作成のリスクを軽減することで検査項目の品質を保証する．

本研究では，検査仕様記述手法を定義するために有限オートマトン (以下 FA) を用いた仕様記述を行なう．CDI ツールの検査対象は構文解析後のソースコー ドであり，検査処理も過去の JCI の開発経験から構文の出現やネスト回数のカウントなどの簡易的な処理であることから，FA で十分モデル化できると判断した．FA を用いて検査仕様を記述するために，構文要素の出現や終了をイベントとして定義し，イベントに対する構文情報の保持や警告条件の判定などの処理をアクションとして整理する．FA のみでは記述出来ない検査に対応するために，スクリプトを提供する．スクリプト記述によってアクションを補完することでより詳細な検査処理記述が可能にする．

本研究では，検査仕様記述を基にした検査処理を実現するために状態遷移機械を用いる．状態遷移機械による検査処理を実現するために JCI の新しいアーキテクチャを設計する．GoF デザインパターン [1] を適用して JCI の検査部と解析部を分離し，仲介するプロバイダを実現することで状態遷移機械を用いた検査を可能にする．検査仕様記述の解析と検査処理コードの生成を行なう自動生成の枠組みを定義し，検査仕様記述を基に状態遷移機械に対応した検査処理コードの自動生成を実現する．最後に，実現した検査処理コード自動生成の枠組みの妥当性について議論する．

2　関連研究

JCI の検査仕様記述方法の提案にあたり，我々は既存のインスペクションツールである CX-Checker[3] の仕様記述方法を調査した．CX-Checker は，ユーザが容易に検査項目の追加・変更可能な言語を持っている．抽象構文木などの情報を XML で表現することで，検査処理論理を XPath 式または DOM 式で記述できる．加えて，検出したい箇所のコード例から，それをマッチさせるための問い合わせ式を導出する機能を備えるなど，ユーザによる検査処理の実現を容易にしている．しかし，CX-Checker は構文解析結果の情報しか持たないので，制御フローを用いた検査の実現について十分な支援ができていない．

3　JCI 概要

JCI は，Java のソースコードに対して静的解析を行ない，欠陥の可能性がある構文や制御フロー，データフローのパターンを発見，指摘するツールである．JCI の処理は，入力されたソースコードから抽象構文木を構築し，その抽象構文木に対して制御フロー解析，データフロー解析を行なうことでフローグラフを構築する．各検査処理は，ソースコードに対する解析結果の抽象構文木やフローグラフを走査し，欠陥の可能性がある箇所を指摘する．

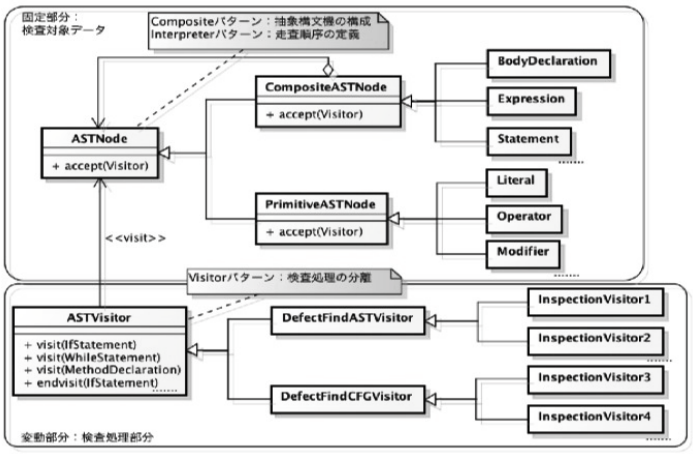

図 1　JCI のアーキテクチャ

JCI は変更に対して柔軟に対応できるよう，固定部分のデータ構造と変動部分の検査処理に分けてアーキテクチャ設計されている．JCI のアーキテクチャを図**??**に示す．抽象構文木の再帰的な静的構造は，Composite パターンを適用している．各構文要素に対する走査処理は Interpreter パターンを適用し，CompositeASTNode クラスにその子要素に対する走査順序を集約することで，検査実行時に必要な走査処理の再利用を行なっている．検査処理の実現には，Visitor パターンを適用する．抽象構文木やフロー

グラフに対する走査順序は Interpreter パターンによって実現されているので，検査処理のアルゴリズムを構文要素から分離して Visitor クラスに集約している．また，検査処理の追加や変更を，構文要素や他の検査処理の振る舞いに影響を与えずに行なうことができることから，保守性や追加変更に対する柔軟性を確保している．

4 JCI の検査処理の変更

検査処理コードの自動生成を実現するために，JCI の検査処理に変更を加える．解析部と検査部間の処理を MessageServiceProvider を仲介して行なうことによって，FA による検査を実現する．新しい JCI の検査処理の概要を図 3 に示す．

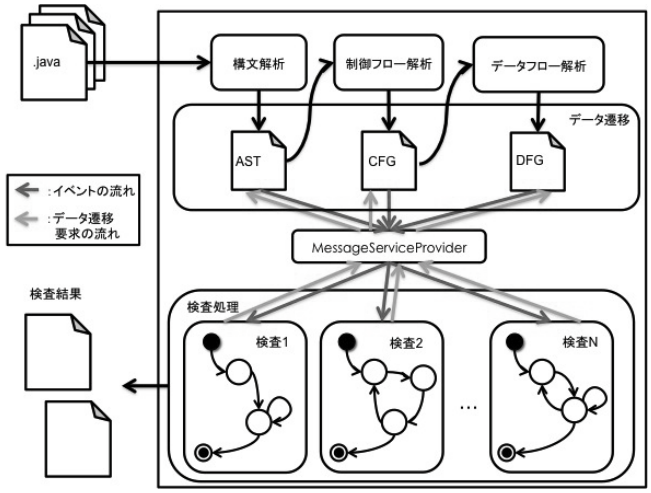

図 2 新しい JCI の検査処理の概要

図 3 の FA で記述した検査処理は MessageServiceProvider を仲介し，抽象構文木、制御フローグラフ、およびデータフローグラフに対してデータの要求を行なう．一方で抽象構文木とフローグラフは遷移したノードをイベントとして MessageServiceProvider に登録し，FA の遷移を発生させる．解析部と検査部は分離されているので，FA に記述するイベントやアクションの変更が解析部に影響することはなく，FA の変更による検査処理の変更が可能になる．

5 検査項目の仕様記述

自然言語と検出例で記述された検査仕様と基に FA とスクリプトを用いて検査仕様の記述を行なう．

5.1 FA を用いた仕様記述

現在の JCI で検査処理を実現する際には，抽象 Visitor のサブクラスに構文要素の種類に応じた検査処理内容を記述する必要がある．構文要素の情報やフロー解析の結果情報は各構文要素が保持しており，それらの利用方法を把握していなければ検査処理の実現は難しい．JCI は，Visitor による行きがけ順走査や帰りがけ順走査で構文要素の出現や終了に着目している．検査仕様を FA で記述するにあたり，既存の JCI の 35 の検査項目を調査し，FA で記述する際に必要なイベント，アクションの調査を行なった．構文要素の出現と終端，走査の終了をイベントとして定義し，アクションとしては構文情報の保持と破棄，警告に関わる判定処理を定義することで FA による仕様記述が実現できる．

5.1.1 抽象構文木に対する検査仕様記述

抽象構文木に関する検査とは，プログラムの構造上に現れる問題の調査を意味する．CDI ツールの検査対象は構文解析後のソースコードであるので，前処理で構文解析は完全であると仮定できることから，FA で十分にモデル化できると考えた．抽象構文木を走査し，走査したノードの出現をイベントとして扱う状態遷移モデルとしてモデル化できる．この検査では，特定のイベントによって検査処理をおこない，特定の条件を満たした場合に警告する．抽象構文木に対する検査項目の例として,「後置式を含む制御文の条件式を検出」がある．この検査項目は，制御文の条件式に後置式が存在した場合に警告する．

この検査項目の検査仕様記述を図 3 に示す．

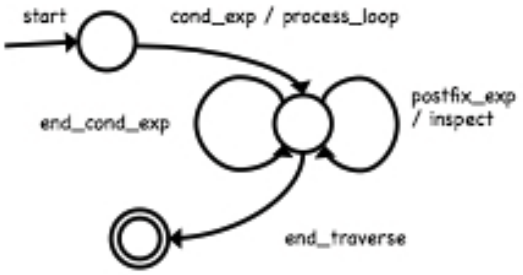

図 3 後置式を含む制御文の条件式を検出

この検査項目の警告例を次に示す．

警告例

```
if(i != 0)
 if(true || i++ < 0){ // 警告
  ...    }
```

この警告例では Search 状態の時に if 文が出現したので，条件式をイベントとして文を保持し遷移する．条件式内の式は中置式であり，後置式が存在しないので再帰処理を行なう. 次の if 文で再度チェックし，後置式が存在するので警告する．

5.1.2 フローグラフに対する検査仕様記述

データフローグラフに対するイベントの生成には，データフローを辿る Visitor を活用し，フローグラフを走査する必要がある．制御フローグラフの各ノードを辿る際，構文要素の種類ごとに変数に着目し，変数の使用点や定義点が存在すればこれらをイベントとして生成する．データフローに対する検査項目の例として.「メソッド仮引数の値を上書きする箇所の検出」がある．この検査処理はデータフローを対象としており，メソッド仮引数の変数がメソッド内で上書きされる場合に警告する．

この検査項目の検査仕様記述を図 4 に示す．

この検査項目の警告例を次に示す．

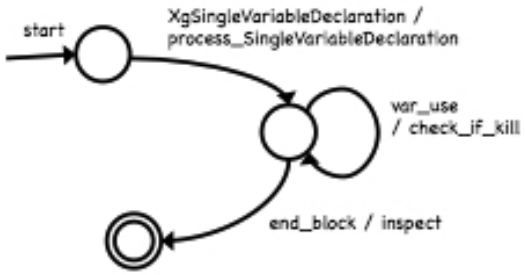

図 4 メソッド仮引数の値を上書きする箇所の検出

警告例

```
public void method(String str){
  if(str.equals(""))
    return;
  str = null; // 警告  }
```

この警告例では，パラメータ宣言文が出現したので，宣言文の情報を保持し遷移する．フローグラフを走査した結果，その後変数の定義点が存在するので代入文を保持する．データフローグラフ終了イベントで警告を行ない，検査を終了する

5.2 アクションを補完するためのスクリプト記述

既存の JCI の検査項目を FA 記述していく中で，FA では記述出来ない検査が存在した．判定に用いる数値の指定や走査結果の保持や利用などの処理である．ループ文の深すぎるネスト構造を検出する検査の仕様記述を図 5 に示す．しかし FA だけではネスト回数のカウントや一定値以上での警告処理を記述することはできない．このような検査を記述するためにスクリプト記述を用いる．

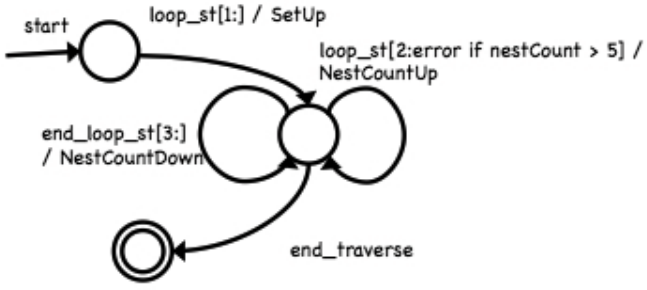

図 5 深すぎるネスト構造を持つループ文を検出

図 5 の検査に対応したスクリプト記述を次に示す．

スクリプト記述

```
Start
int nestCount
SetUp(){ nestCount = 0; }
NestCountUp(){ nestCount ++;
   if(nestCount > 5){ warnning(); }}
NestCountDown(){ nestCount --; }
end
```

各アクションに対して引数と処理内容を記述する．状態遷移機械の遷移に応じてスクリプトが呼び出され，イベントに対応する処理を実行する．上記の例では nestCount を加算，減算しネスト回数が 5 回を超えたら警告を行なう．このようにスクリプト記述を用いることでアクションをより柔軟に利用することが可能である．

6 JCI の新しいアーキテクチャ

状態遷移機械を用いた検査を実現するために既存の JCI のアーキテクチャに変更を加える．

変更後のアーキテクチャを図 6 に示す．

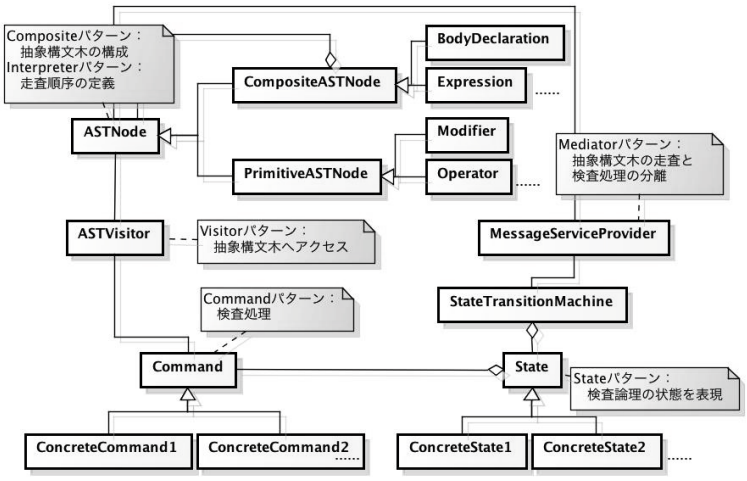

図 6 新しい JCI のアーキテクチャ

既存の JCI のアーキテクチャからの変更点は State パターンと Command パターン，Mediator パターンを新たに適用したことである．抽象構文木を構築する Composite パターンと Interpreter パターンを変更する必要はないが，Interpreter パターンにはイベントの遷移を状態遷移機械に通知する処理を追加する．Visitor パターンは既存の JCI では抽象構文木，フローグラフなどのデータ構造に対するアクセスと，その結果を利用した検査処理の実現に用いてきたが，新たな JCI ではデータ構造に対するアクセスのみに利用する．検査処理については状態遷移機械を用いて実現するために，State パターンと Command パターンを適用する．State パターンで状態とその遷移を定義し，状態の遷移に対する振舞いは Command パターンを用いて定義する．JCI のデータ構造と状態遷移機械による検査処理を仲介し，状態遷移機械の追加を容易にするために，Mediator パターンを用いる．MessageServiceProvider は解析部からの状態遷移機械へのイベント遷移に対する要求と，状態遷移機械からのデータ構造の走査に対する要求を仲介する．これにより，状態遷移機械による検査処理の実現が可能になる．

7 考察

7.1 検査処理コードの自動生成に関する考察

検査仕様に着目したい構文要素や着目した構文要素に対する振舞いを明確に記述することで，検査処理コード

の自動生成が可能になると考える．検査仕様は，検査処理論理を FA で記述している．FA は Visitor によって走査処理された構文要素の出現や終了をイベントとし，検査処理論理の走査情報を取得する．各状態とイベントによる遷移は State パターンを用いることで実現し，FA におけるアクションを Command パターンを用いて定義することで追加・変更を容易にする．各モデルから得られる状態とイベント送受信の実行順序をシーケンス図で記述し，アクションに対する詳細な要求をスクリプトを用いて記述する．検査仕様の各アクションとスクリプトには処理コードを格納する．検査仕様のイベントとアクションに対応した処理コードの組み合わせ，定義した実行順序に沿って生成することで検査処理コードを実現できる．

検査処理コード生成の手順を図 7 に示す．

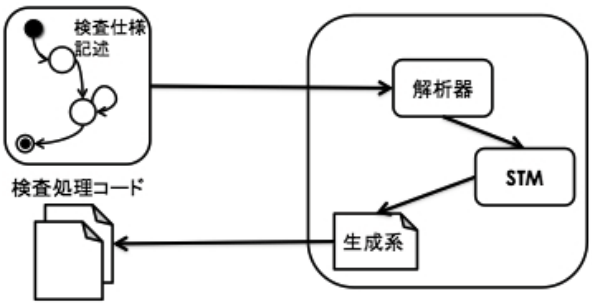

図 7 検査処理コード生成手順

入力された状態遷移モデルとスクリプト記述を解析器で解析し，State パターンと Command パターンを用いた状態遷移機械へと変換する．そして各状態とイベントアクション，スクリプトに対応する処理コードを生成することで検査処理コードの自動生成を実現する．

7.2 構文要素のパラメータ化

類似した処理の対象となる複数の構文要素を一つのパラメータとしてまとめる．アクションはパラメータ化した要素を受け取り処理する．こうすることで類似した機能を持つアクションを構文要素ごとに作成する手間を省略できる．例えば If 文や While 文，For 文はともに条件式に対する検査の対象になる構文要素である．これらの構文要素は Process_loop アクションのように制御文を収集するアクションのパラメータとなる．これらの構文要素を「条件式を持つ文」としてパラメータ化し，Process_loop アクションで処理する．例として図 5 の検査仕様の loopst は For 文，拡張 For 文，While 文，DoWhile 文などをパラメータ化し一つのイベントの発生としてまとめたものである．

既存の JCI の検査処理を調査した結果，パラメータ化できた構文要素の一部を表 1 に示す．

検査の仕様と実行するアクションによって検査の対象となる構文要素は多様であり，個々の構文要素に対応した様々なアクションが存在する．今後は検査項目の追加・変更にともなってさらに，構文要素のパラメータ化を行なっていく必要がある．

表 1 パラメータ化する構文要素

パラメータ名	構文要素
繰り返し文	While 文，DoWhile 文，For 文
分岐文	If 文，Switch 文
条件式を持つ文	If 文,While 文,DoWhile 文,For 文
中断文	Break 文,Return 文,Continue 文
変数宣言文	変数宣言, フィールド宣言,…

7.3 処理コードの再利用

各アクションは既存の JCI の検査処理コードを再利用する．再利用する際にはアクションの持つ機能によって，コードを選別し整理する必要がある．JCI の検査処理は検査の対象となる構文要素ごとに処理が記述されているが，構文要素ごとにその都度アクションを作成すると検査仕様記述が煩雑になる．これに対して構文要素をパラメータ化し，統一できる処理を一つのアクションにまとめる．これによりアクションと構文要素は 1 対 1 で対応するのではなくて，同様の条件や要素を持つ複数の構文要素に対してアクションを利用できる．

例として図 3 の Process_loop アクションは if 文に限らず，条件式を持つ For 文などの他の文に対しても利用できる．inspect アクションも同様に条件式の親となる文にかかわらず，全ての条件式に対して判定することができる．

8 おわりに

本研究では，既存の検査項目の開発事例から仕様記述に必要な情報を整理し，検査仕様記述方法の提案を行なった．検査仕様を定義したことにより，異なるデータ構造でも統一的に検査仕様を記述できた．検査仕様記述から検査項目を生成するアーキテクチャを設計した．今後の課題として，処理コード自動生成による検査作成コスト削減と誤りリスク低下の評価が挙げられる．また，多くの検証による信頼性の向上と検査仕様を記述するための，開発補助インターフェースの構築を行なう必要がある．

参考文献

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

[2] 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.

[3] 大須賀俊憲，小林隆志，間瀬順一，渥美紀寿，山本晋一郎，鈴村延保，阿草清滋，“CX-Checker: C 言語プログラムのためのカスタマイズ可能なコーディングチェッカ,”ソフトウェア工学最前線 *2009 (*情報処理学会ソフトウェアエンジニアリングシンポジウム *2009* 論文集*)*, 近代科学社, pp.119-126, 2009.