CDIツールの検査項目の作成支援に関する研究

2008MI033 早川 顕司　　2008MI063 今村 匡利　　2008MI064 井上 兼斗
指導教員　張 漢明

1　はじめに

コードインスペクションとは，ソースコードの品質を向上させる目的として，プログラムなどの成果物を実際に動作させることなく，誤りや不具合があるかを検証する作業である．本研究室では，Java のソースコードに対して，コードインスペクションツールである Java Code Inspector[3](以下 JCI) を開発している．JCI は入力されたソースコードに対して構文解析やフロー解析などを行ない，その結果を基にプログラムの欠陥の可能性となる箇所を検出する．現在，JCI に新たな検査項目を追加しようとした場合，開発者が仕様を自然言語で定義し，それを基にプログラムを作成している．プログラムを作成する上で，開発者は JCI の内部構造に精通している必要があり，内部構造を理解していない人が書き直すのは困難である．したがって，内部構造を理解しなくても検査項目の作成ができる仕組みが必要である．

本研究の目的は Java の構文知識に基づいて新しい検査項目の作成を支援する環境を提供することである．目的を達成するために，構文解析木の情報を基にコードインスペクションを行なう手法を提案する．そのために，検査項目を表現する状態遷移機械を定義する．定義した状態遷移機械を基に検査を行なうために，抽象構文木を深さ優先で探索し，探索の各地点において各構文要素の解析の開始や終了というイベントを発生させる．そのイベントに基づいて定義した状態遷移機械上で状態を遷移させることで検査項目を実現する．

本研究では，状態遷移機械を用いて検査項目を実現する手法について記述する．そして検査項目を実現する際に状態遷移機械上で実現する必要があるイベントやアクションについて考察する．提案手法に基づいて解析器を試作し，既存の JCI のアーキテクチャがどう変化したかについて考察する．さらに，本手法を用いて実現できる既存の JCI の検査項目について考察したうえで，今後の拡張の方向性について考察する．

2　背景技術

JCI は入力された Java ソースコードに対して静的解析を行ない，不具合が生じる可能性のある部分を指摘するツールである．図 1 に JCI の概要を示す．JCI はソースコードに対して構文解析を行ない，抽象構文木を生成し，それに対してフロー解析を行ない，制御フローグラフとデータフローグラフを生成する．個々の検査項目は，ソースコードに対する解析結果としての抽象構文木やフローグラフを走査することで，欠陥の可能性となる箇所を検出する．検査項目のソースコードは，検査対象となる構文要素を取得したあとで，条件に応じた判定を行ない，必要に応じて警告する．

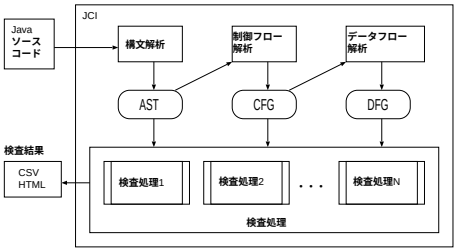

図 1　JCI の概要

3　検査項目の仕様記述

われわれは，抽象構文木の探索によって実現可能な検査項目を対象として仕様を再定義した．検査したいソースコードに対してあらかじめ構文解析を行ない，抽象構文木を生成する．次にこの抽象構文木に対して深さ優先探索を行なうことで，構文要素がどのような順番で出現するかを追跡し，if 文や for 文などの構文要素の開始や終了のイベントを発生させる．

あらかじめそれぞれの検査項目は状態遷移図として表現し，発生したイベントに基づいて状態を遷移させ，検査処理を行なう．全てのイベントを受け取った時点で検査を終了するが，途中で警告状態に遷移したら警告の処理を行なう．Java は LALR(1) 文法であるので，本来ならば文法を受理するためにプッシュダウンオートマトンが必要であるが，検査項目においては特定の非終端記号についてのみ考慮すれば良い．よって，スタックを使用して復帰させる場所を解析中に適宜登録する．検査の流れを図 2 に示す．

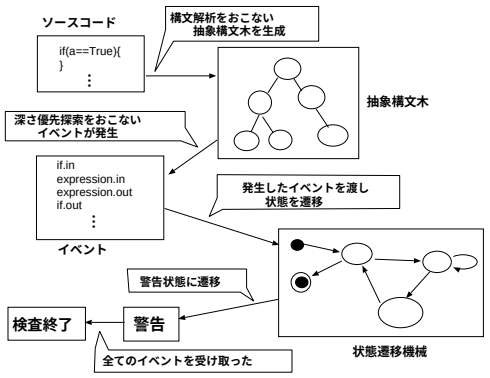

図 2　検査の流れ

4　検査項目の記述方法

検査項目の例として,「if 文の条件式にインクリメント演算子やデクリメント演算子の後置式が存在する」場合に警告する検査項目を挙げる．この検査項目の仕様は，if 文の条件式にインクリメント演算子やデクリメント演算子の後置式が存在する場合に警告を行なうものである．この検査項目の状態遷移図を図 3 に示す．

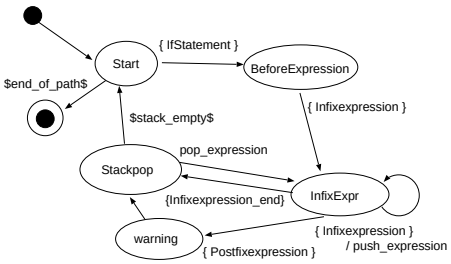

図 3　if 文の条件式内に後置式が存在するか

この検査項目の遷移の流れを以下に示す．

1. 検査対象のソースコード内に if 文が存在するなら，**Start** 状態から **BeforeExpression** 状態に遷移
2. **BeforeExpression** 状態では if 文に式が存在するなら，**InfixExpr** 状態に遷移
3. **InfixExpr** 状態では if 文の式が後置式ならば，**warning** 状態に遷移し警告．式が複数あればスタックに **push** し **InfixExpr** 状態に再帰，着目する式がなければ **Stackpop** 状態に遷移
4. **Stackpop** 状態ではスタック内に式が格納されている場合，式を **pop** し **InfixExpr** 状態に遷移．スタックが空の場合は，**Start** 状態に遷移
5. **Start** 状態で検査対象のソースコード内に存在する全ての if 文に関して検査した場合に終了状態に遷移し検査を終了

5　解析器の試作

5.1　アーキテクチャの変更

4 章で提案した検査項目の記述方法を適用するために，JCI のアーキテクチャの変更をする．現在の JCI のアーキテクチャを図 4 に示す．

現在の JCI のアーキテクチャは変更に対して柔軟に対応できるように，GoF デザインパターン [1] を用いてアーキテクチャが設計されており，データ構造の部分と検査処理の部分に分けられて設計されている．解析器の試作においては，検査処理の部分に手を加える．

現在の JCI のアーキテクチャは，Java のソースコードから抽象構文木を作成する際，抽象構文木の再帰的な構造を表現するために Composite パターンを利用している．Interpreter パターンを利用することで，その抽象

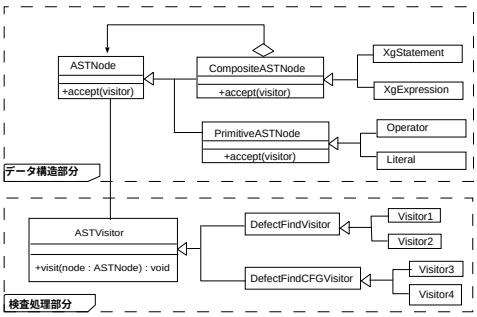

図 4　現在の JCI のアーキテクチャ

構文木の走査順序を定義する．Visitor パターンによって抽象構文木のデータ構造の部分と検査処理が分離されている．検査項目の追加や変更を，構文要素や他の検査処理に影響を与えずに行なうことができる．現在の JCI の検査処理の部分では状態と状態の追加ならびに状態の遷移を表現できないので，現在の JCI のアーキテクチャでは 4 章で記述した検査項目を本研究で提案した手法で表現するように設計されていない．これにより，状態の追加や状態遷移を記述できるようにするために現在の JCI のアーキテクチャを変更する必要がある．

5.2　変更したアーキテクチャの構成

状態の追加や状態遷移の定義をするために，現在の JCI のアーキテクチャに GoF デザインパターンの State パターンと Command パターンを適用した．State パターンと Command パターンを適用した際のアーキテクチャを図 5 に示す．

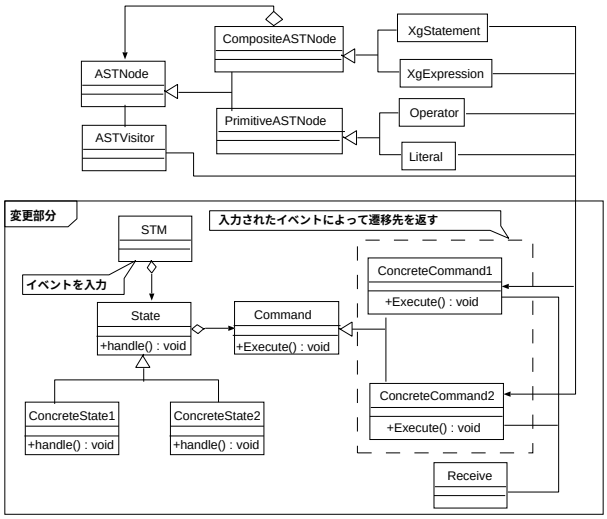

図 5　変更したアーキテクチャ

抽象構文木の再帰的な構造や走査順序は変更していない．図 3 でわれわれが定義した状態遷移図を表現するために，新しく State パターンと Command パターンを追

加することで，検査項目の仕様記述から作成した状態遷移の処理を実現する．State クラスや Command クラスを状態遷移機械に対して追加，変更を加えることで新規の検査項目を容易に実現できる．

5.2.1 State パターン

State パターンは，状態をクラスとして表現し，状態ごとに振る舞いを切り替えることができるパターンである．State パターンを適用すると，それぞれの状態が独立したクラスとして実装できる．その結果，新しい状態を追加する場合には，新しい状態のクラスを作成するだけで済むので，他のクラスに対して影響を与えずに修正や変更，追加をすることができる．本研究では，状態遷移図の各状態をクラスとして表現する．4 章の状態遷移図から Java の State クラスの構造を図 6 に示す．

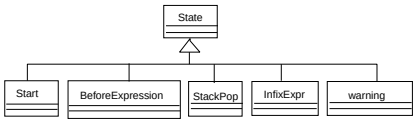

図 6　State クラスの構造

図 6 の State クラスは抽象クラスであり，それを実現する各クラスは図 3 に示している各状態に対応している．新たに図に状態を追加した場合，それに応じて新たなクラスを作成する．

5.2.2 Command パターン

Command パターンは，命令をクラスとして表現することができるパターンである．Command パターンを適用すると，命令の受け付けと命令に対応する処理を切り離して実装することができる．その結果，新しい命令が追加された場合には，既存の命令のクラスを修正する必要がない．また追加された命令に対応する処理のクラスを作成するだけで済み，新しい命令を追加する場合でも，影響を与えずに命令の修正や変更，追加を容易に行なうことができる．本研究では，状態遷移図のある状態から遷移先の状態への遷移を表現できる．4 章の状態遷移図から Java の Command のクラスの図 7 を作成した．

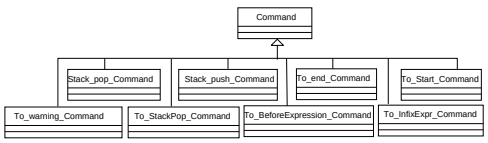

図 7　Command クラスの構造

図 7 の Command クラスは抽象クラスであり，それを実現する各クラスは図 3 に示している任意の状態から別の状態への遷移と対応している．新たに図に遷移を追加したい場合，それに応じて新たな状態に対応するクラスが必要である．

5.3 試作した解析器

解析器の例として 4 章で作成した状態遷移図を用いて検査項目を作成した．この検査項目の警告例を以下に示す．

警告例

```
1: public static void main(String[] args){
2:   int i = 0;
3:   if(i == 0){
4:    if(i-- != 1){  // 警告
5: } } }
```

警告例のイベントの発生順序を以下に示す．

1. 1，2 行目では遷移に関係するイベントは発生しない
2. 3 行目で if 文があるので，イベント **IfStatement** が発生する
3. 3 行目の if 文の条件式内に中置式 i == 0 があるので，イベント **Infixexpression** が発生する
4. 3 行目に中置式は i==0 以外には無いので，イベント **Infixexpression_end** が発生する
5. スタックに push するイベントが発生していないので，スタックが空であることを表現するイベント **stack_empty** が発生する
6. 4 行目で if 文があるので，イベント **IfStatement** が発生する
7. 4 行目の if 文の条件式内に中置式 i - - != 1 があるので，イベント **Infixexpression** が発生する
8. 中置式 i - - != 1 内に後置式 i - -があるので，イベント **Postfixexpression** が発生する
9. スタックに push するイベントが発生していないので，スタックが空であることを表現するイベント **stack_empty** が発生する
10. 5 行目では遷移に関係するイベントは発生しない
11. ソースコードの探索が終わったので，イベント **end_of_path** が発生する

以上の発生したイベントを状態遷移機械に渡し，遷移させた結果，試作した解析器で警告を行なうことができた．

6 考察

6.1 生成系に関する考察

現在の JCI の開発者は自然言語で書かれた仕様を基に，検査項目を作成している．本研究では状態遷移機械を用いて検査項目を実現する手法として，記述された仕様から状態遷移図を作成する手法をあげた．しかし，現在は Java のソースコードへの変換は手作業で行なっている．この作業を支援するための方法として二つの方法が考えられる．一つ目の方法は，状態遷移図から Java のソースコードを自動生成することである．二つ目の方法は Metal[2] のような仕様記述言語から状態遷移を自動生成することである．これらの二つの方法を比較し自動生成ができる環境を整備することが求められる．

6.2 既存の検査項目に対する分析の考察

われわれは，状態遷移図を作成するために検査項目の分析を行なった．分析の基準として，JCI の検査項目が検査対象となるソースコードに対してどのような構文要素に着目し，着目した際に各構文要素の解析の開始や終了の処理をどのように行なうかを分析した．既存の検査項目 36 項目を分析した結果，36 項目の内，9 項目が検査対象の構文要素がプログラム内にあるかどうかを検査する検査項目である．本研究ではこれらの 9 項目の検査項目を対象とした．本研究で対象としない事例の分類を行なった結果，以下のように分類ができた．

- カウントを扱う検査項目
- 定数伝播を扱う検査項目
- オブジェクト伝播を扱う検査項目
- 制御フロー，データフローを扱う検査項目
- リストを扱う検査項目
- メソッド間の分析を扱う検査項目

カウントを扱う検査項目の分類は，検査対象がネスト構造になっているものを検出する場合の検査項目である．定数伝播，オブジェクト伝播を扱う検査項目の分類は，検査項目内で定数伝播，オブジェクト伝播の結果を用いることで検査を行なっている検査項目である．制御フロー，データフローを扱う検査項目の分類は，検査項目内で抽象構文木だけでなくフローグラフを利用している検査項目である．リストを扱う検査項目の分類は，検査項目内でリスト構造を用いて検査項目を作成している検査項目である．メソッド間の分析を扱う検査項目は，検査項目内でメソッド間で解析をすることにより検査を行なっている検査項目である．

36 項目の検査項目を分類した結果を表 1 に示す．また検査項目の中にはメソッド間分析とリストを使う等重複している検査項目も存在している．

表 1　検査項目の分類

パターン	要素の存在	カウント
検査項目の数	9	3
パターン	定数伝播	オブジェクト伝播
検査項目の数	8	5
パターン	CFG・DFG	リスト
検査項目の数	6	7
パターン	メソッド間の分析	
検査項目の数	1	

既存の検査項目 36 項目のうち 9 項目の検査項目に対して状態遷移図の作成を行ない，状態遷移図から新しい検査項目を作成することができた．この 9 項目の検査項目は抽象構文木を辿るだけで検査することができる検査項目で，「要素の存在」を表現している検査項目である．残りの 27 項目の検査項目は，単純に抽象構文木を辿るだけでは検査することができない．データフロー，制御フローを扱う検査項目はフローグラフを辿る方法を考慮する必要がある．定数伝播，オブジェクト伝播を扱う検査項目は定数伝播，オブジェクト伝播の特徴と定数伝播，オブジェクト伝播を行なった結果の取得を状態遷移図で表現することで検査項目を作成できると考える．リストを扱う検査項目は，リストの構造を状態遷移図で表現できれば検査項目を作成できる．メソッド間分析を扱う検査項目は，メソッドとメソッドの間で行なわれるデータの受け渡しを状態遷移図で記述できれば検査項目の作成ができる．よって，今後は残りの 27 項目の検査項目に対して状態遷移図を記述する必要がある．また本研究の手法を用いて状態遷移図から検査項目の試作を行なう．そして，自動生成できる環境を提供する必要がある．

7 おわりに

本研究では，検査項目の仕様記述から状態遷移図を作成できた．そして，警告例を使って検査し，本研究の手法で警告できる新しい検査項目の処理の作成を行なうことができた．また現在の JCI のアーキテクチャに State パターンと Command パターンを追加することで JCI で状態遷移図の各状態と遷移先の状態への遷移を表現することにより検査項目の状態遷移を記述することができるアーキテクチャの作成ができた．

今後の課題として，仕様記述言語 Metal や状態遷移図から生成するツールを使用することで自動生成できる環境を整備すること．抽象構文木を単純に辿る検査項目だけでなく，データフローや制御フロー等を扱う検査項目ごとの特徴を考慮する必要がある検査項目の仕様を作成すること．状態遷移機械を作成し検査項目の試作を行なうこと．また，既存のアーキテクチャに State パターンと Command パターンを適用したが，他のデザインパターンとの比較を行ない，妥当性を考えることが必要である．

参考文献

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

[2] S. Hallem, B. Chelf, Y. Xie, and D. Engler, “A System and Language for Building System-Specific, Static Analyses,” *PLDI ’02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 69-82, 2002.

[3] 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，” ソフトウェア工学の基礎 XVII (日本ソフトウェア科学会 FOSE2010)，pp.15-24，2010.