コード検査ツール開発における検査仕様記述の提案とテストケースの自動生成に関する研究

M2011MM034 加藤遼介
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，ソースコードを静的に検査し，ソフトウェアの潜在的な欠陥を検出するツールである．様々なプログラミング言語に対して CDI ツールが開発され，実開発プロジェクトに適用されてきた [4]．CDI ツールの検査に対する要求はプロジェクトごとに異なる．一方，多くの CDI ツールは検査のカスタマイズ機能を備えていないか，その機能が限定的である．

本研究室では Java を対象とする CDI ツール（以下, JCI[3]）を開発している．JCI のアーキテクチャは GoF デザインパターン [2] に基づき検査処理の追加や変更に対する柔軟性を重視して設計されている．特に検査処理には Visitor パターンが適用されており，ツール開発者は Visitor の変更することで，様々な検査を実現できるのでカスタマイズ性は高い．一方で，ツール利用者によるカスタマイズは JCI の内部構造に習熟していなければ困難である．また，検査仕様は自然言語と警告例をもとに記述されており，あいまいさを含む．あいまいさは複数の解釈を生むことから，誤った実現を引き起こす可能性がある．さらに，あいまいな仕様をもとに適切なテストケースを作成することは困難であるので，ツール利用者にカスタマイズされた検査の品質の保証もまた困難である．

本研究の目的はテストケースの自動生成による系統的品質保証である．系統的品質保証は，検査仕様の可視化，仕様に対する適切なテストケースの生成，生成されたテストケースを用いてテストし，仕様の実現に誤りがないことの保証という 3 点からなる．本研究では，有限オートマトン（以下，FA）により検査仕様を可視化する．そしてテストケース自動生成系を構築することで，テスト漏れの排除し，実現した検査の品質を保証できる．

言語処理はトークンの出現をイベントと捉えた時に，それを処理するオートマトンとしてモデル化できる．例として構文解析はプッシュダウン・オートマトンとしてモデル化できる．FA では文脈自由文法であるプログラミング言語の構文解析に関する処理のモデル化はできない．しかし，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 の開発経験からネストの出現のカウントなどの簡易的な処理であることが分かったので，正規文法相当の FA で十分モデル化できると判断した．FA でモデル化できない副作用を伴なう検査仕様についても，カウント操作などの単純な処理であるので，スクリプトにより FA のアクションを記述できる．全体として，FA とスクリプトによる記述で検査仕様を可視化できる．テストケースの自動生成は FA にパターンがあることに着目し，パターンと出力されるコード片を対応付けることで実現する．

2 関連研究

2.1 CheckStyle

CheckStyle[1] とは，Java を対象とするオープンソースの CDI ツールである．CheckStyle はあらかじめ提供された検査以外に，検査のカスタマイズが可能である．カスタマイズは 1) GUI を用いた方法，2) XML を用いた方法，3) API を利用する Java プログラムを用いた方法の 3 種類の方法によって可能であるが，1) 主にコーディングスタイルの検査にとどまっている，2) 前者 2 つのカスタマイズは文字数制限などの数値の変更が主であり限定的である，3) 後者は API を直接利用したプログラム記述が必要でありツール利用者の労力が大きいという理由から，ツール利用者による実用的なカスタマイズは困難である．

2.2 CX-Checker

CX-Checker[5] とは，C 言語を対象とする CDI ツールであり，ソースコードを入力として検査をおこなう．CX-Checker は検査機能のカスタマイズを重視して構築されており，カスタマイズには 1) XPath 式を用いた方法，2) DOM 式を用いた方法，3) Java プログラムによるラッパーを用いた方法の 3 種類がある．カスタマイズには C 言語の抽象構文木を XML で表現した CX-Model に習熟している必要があるが，CX-Model に習熟していないツール利用者のための補助インターフェースを備えており，カスタマイズが容易である．一方でフローグラフに関する検査の実現のための支援機能が十分ではないことから，ツール利用者による実用的なカスタマイズは困難である．

3 検査項目の仕様記述

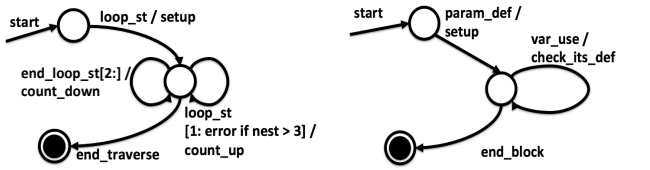
3.1 状態遷移モデルを用いた検査仕様記述

本研究では，検査処理の対象が構文解析後のソースコードであることに着目し，正規文法相当の FA である状態遷移モデルを用いて検査仕様を記述する．CDI ツールがおこなう検査は着目する視点ごとに抽象構文木の構造に対する検査，変数の宣言と使用に関する検査，制御フローグラフに関する検査に分類される．いずれの検査も状態遷移モデルで記述できることを確認する必要がある．

3.1.1 抽象構文木の構造に関する検査仕様記述

抽象構文木の構造に関する検査とは，プログラムの構造上に現れる問題の調査を意味する．文脈自由文法であるプログラミング言語は，プッシュダウン・オートマトンで処理される．しかし，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．検査処理はカウントなど簡易的な検査処理で十分であり，正規文法相当の FA で十分に

モデル化できると考えた．具体的には，この検査は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．



(a) 抽象構文木の構造に関する検査　(b) 変数の宣言と使用に関する検査仕様

図 1　検査仕様の記述例

図 1(a) に検査仕様の記述例を示す．この検査仕様は，3 回以上ネストしているループ文があるかを検査する．イベントはループを構成する文の出現を表わす loop_st，ループを構成する文の終了を表わす end_loop_st，構文木の走査の終了を表わす end_traverse の 3 つである．アクションはカウントの初期化をおこなう setup，カウントを 1 増やす count_up，カウントを 1 減らす count_down の 3 つである．ガード条件の”数字:”は遷移順序を表わし，”error if nest $>$ 3”は警告する条件を表わす．この検査では構文木を走査し，ループ文の出現に応じてカウントを走査する．そして，カウントが 3 を越えた場合に警告する．

3.1.2　変数の宣言と使用に関する検査仕様記述

変数の宣言と使用に関する検査とは，変数の不正な定義と利用の関係の調査を意味する．これは，抽象構文木の各点における到達定義を調べることで実現できる．到達定義は，前処理で抽象構文木に対するフローグラフ解析がおこなわれているので，構文木の各点においてその計算結果を利用すればよい．したがって，この検査は抽象構文木の構造と同様に FA でモデル化が可能である．

図 1(b) に検査仕様の記述例を示す．この検査は引数が上書きされているかを検査する．イベントは引数の行き掛け順の走査を表わす param_def，引数の使用点を表わす var_use，引数の有効範囲を抜けたことを表わす end_block の 3 つである．アクションは引数の初期情報を取得する setup，使用点における到達定義を取得する check_its_def の 2 つである．この検査では，setup で取得した情報と check_its_def で取得した到達定義を比較し，到達定義が異なる場合は値が上書きされたとみなし，警告する．

3.1.3　制御フローに関する検査仕様記述

制御フローに関する検査とは，プログラムの制御構造上の問題の調査を意味する．例として無限ループと到達しないコードに分類される．この検査は制御フローを走査し，出力辺がないノードを警告することで実現できる．しかし，CDI ツールでおこなう検査では，制御構文を構築するノードの条件式の真偽や含まれている分岐を表す文のみに着目すればよい．したがって，この検査は抽象構文木の構造に関する検査と同様であるとみなせる．

3.1.4　スクリプトによるアクションの検査仕様記述

状態遷移モデルを用いることで，CDI ツールに求められる検査仕様を記述できた．しかし，状態遷移モデルではモデル化できない仕様が存在する．それはカウントや値の取得などの副作用を伴なう処理の扱いである．3.1.1 節の図 1(a) の検査仕様ではカウントを表す値がそれに該当する．状態遷移モデルでこのような値を扱う場合，そのモデルが情報を保持する必要がある．ツール利用者によるカスタマイズを実現する場合，どのような値が利用されるかは検査により異なるので，検査のカスタマイズの度にモデルが持つ情報もカスタマイズされる必要がある．モデルの内部構造が変更された場合，他の検査にも影響が及ぶことが予想される．その結果，検査の品質が損なわれる可能性がある．したがって，本研究では副作用を伴なう処理を記述するためのスクリプトを用いてモデルのアクションを記述した．スクリプトは STM の遷移に応じて参照され，イベントに対する処理をおこなう．

図 1(a) に対するスクリプト記述例

```
global count;
setup(){ count = 1; }
count_down(){ count = count - 1; }
count_up(){ count = count + 1;
            Ensures(count < 3)}
```

図 1(a) に対するスクリプト記述例を上記に挙げる．global 修飾子は変数がアクション間で共通に利用されることを意味する．アクションは”アクション名 (引数){ 処理内容 }”という形で記述される．STM の状態遷移に応じてスクリプトが呼び出され，対応する処理を実行する．この例ではアクションの呼び出しに応じて loop_st の出現回数が操作される．Ensures は事後条件を意味し，事後条件を満たさない場合，対応するイベントを警告する．

4　状態遷移モデルによる検査を実現するアーキテクチャ

本研究で提案し検査仕様記述による検査を実現する CDI ツールのアーキテクチャを図 2 に示す．

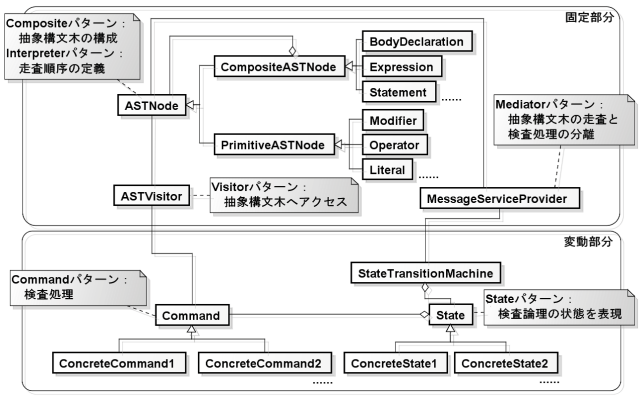


図 2　CDI ツールのアーキテクチャ

抽象構文木の構造は Composite パターンによって実現

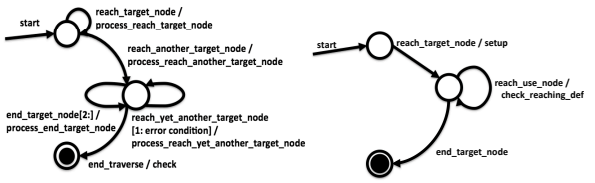
する．フローグラフはこの構文木にまたがるグラフとして実現される．その走査と走査したノードの通知処理は Interpreter パターンを用いて実現する．さらにツール利用者から構文木のデータ構造を隠蔽化するために Visitor パターンを適用した．状態遷移モデルとスクリプトを表現するために，State パターンと Command パターンをそれぞれ適用した．State は状態を，Command は遷移に伴なうアクションを表現する．抽象構文木と状態遷移モデルのイベントの通信を実現するために，Mediator パターンを適用した．Mediator パターンにより抽象構文木やフローグラフに対する走査の要求と STM に対する遷移の要求とを仲介することで，走査処理と検査処理を分離する．

5 テストケースの自動生成の枠組みの構築

テストケースの自動生成を実現するために，検査仕様の状態遷移モデルにはパターンがあることに着目した．パターンを定義し，パターンと出力されるコード片の対応関係を定義する．そして対応関係とアクションの組み合わせによってテストケースを自動生成する．

5.1 抽象構文木の構造に関する検査

本研究で提案する検査仕様記述を用いて JCI の検査仕様を記述した結果，1 つの状態遷移モデルにつき警告対象のノードは 1 つだけであるという制限を与えれば，全ての検査対象は「ある構文要素の中の特定の構文要素の出現」という形で記述できることが分かった．そして，それに対する状態遷移モデルも全て図 3 の形で記述できると考えた．これをパターンとし，そのパターンに対しコード片を適切に対応付けられれば，テストケースを自動生成できると考える．



(a) 抽象構文木の構造に関する検査 (b) 変数の宣言と利用に関する検査

図 3 検査仕様のパターン

抽象構文木に関する検査仕様は全て図 3(a) の形で記述できる．このパターンでは，start の遷移に接続される状態は 0 回以上の自己遷移を持ち，end_traverse に接続される状態は 2 回以上の自己遷移を持つ．この自己遷移は検査で着目したい要素の数によって増減する．アクションとガード条件の内容や有無は検査によって異なる．

今回は図 1(a) を例として自動生成をおこなう．図 4 は検査仕様に対する遷移順序とコード片を記述したものである．スクリプトは 3.1.4 節で示したものとする．テストケースを生成するために，全ての遷移について 1 回ずつ遷移する．その際，同時に 2 つの遷移が起こりうる場合は，ガード条件に記述した遷移順にしたがって遷移する．この例では loop_st，loop_st，end_loop_st，end_traverse の遷移が起こる経路が得られる．ここから，遷移に対応

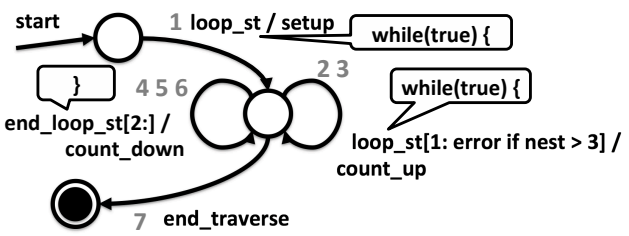


図 4 テストケース生成例

するコード片を出力することで，テストケースが得られる．さらに，より検査仕様に即したテストケースを得るためにスクリプトを参照し，経路情報に反映する．スクリプトのうち，テストケース生成にかかわる処理はカウントの操作，ノード情報の取得，取得したノード情報の比較の 3 つである．カウントであれば規定回数遷移する経路を得る．後者 2 つであれば，比較対象の情報をもとに具体的な値をコード片に反映させる．今回は事後条件を参考に loop_st と end_loop_st の回数が経路情報に反映される．その結果，loop_st が 3 回，end_loop_st が 3 回，end_traverse という経路が得られる．これらのコードが全て同一のメソッド内にあると仮定すれば，loop_st に対して”while(true) {”，end_loop_st に対して”}”のコード片を出力することで，以下のようなテストケースを得ることができる．

得られるテストケース例

```
1: public class test {
2:       public static void main(String args[]) {
3:           while(true) {
4:               while(true) {
5:                   while(true) { } } } } }
```

5.2 変数の宣言と使用に関する検査

変数の宣言と使用に関する検査のパターンも 5.1 節の図 3(b) のように定義できる．このパターンでは，start の遷移に接続される状態は 0 回の自己遷移を持ち，end_traverse に接続される状態は 1 回以上の自己遷移を持つ．抽象構文木の構造に関する検査と異なるのは，フィールド変数など，様々なメソッドやクラス間でまたがって利用される変数の扱いである．本研究では，この検査に対するテストケースは変数の定義点と使用点が全て同一のメソッド内にあると仮定する．同一のメソッド内に出力すると考えれば，テストケースは 5.1 節と同様の手順で得られる．

5.3 制御フローに関する検査

3.1.3 節で述べたように，この検査は抽象構文木上に現れる制御構文を表す構文要素の条件式の真偽や分岐を表す文に着目すれば実現できるので，抽象構文木の構造に関する検査と同様に記述できる．したがって，この検査のパターンとそれに対するテストケースも同様に得られる．テストケースを生成する際に抽象構文木の構造に関する検査と異なる点は，この検査では構造上に現れない条件

式の真偽が重要である点である．これは条件式に対してスクリプトを参照し，情報を反映することで対処する．

6 考察

6.1 状態遷移モデルを用いた検査仕様記述に関する考察

本研究では，状態遷移モデルを用いて検査仕様を可視化した．そして状態遷移モデルを用いて JCI が備える検査仕様を全て記述できることを確認した．したがって，本研究で提案した検査仕様記述は妥当であると考える．しかし，テストケースの自動生成の際には，構文の出現箇所の特定と警告対象の除外の実現という 2 つの課題がある．

構文の出現箇所の特定とは，複数のメソッドやクラスにまたがって現れうる構文の出現箇所の特定が困難であるという問題である．例としてフィールド変数が挙げられる．フィールド変数はクラス内の複数のメソッドで利用される可能性がある．検査は構文木を走査して現れるイベントに反応する STM を用いて実現できる．テストケースを生成する場合は，イベント列から適切な木構造を得られないという問題がある．すなわち，複数の定義点や使用点の出現に対して，単一のメソッド内に出力するか，複数のメソッド内へ出力するか特定できない．この問題に対する対処として，複数の使用点や定義点に対して検査処理がおこなわれる場合，単一のメソッドか複数のメソッドか，どの出力をおこなうかツール利用者に提案することが挙げられる．検査に対する要求の明確化により，要求に即したテストケースを自動生成できる．

警告対象の除外機能とは警告対象となる項目のうち，特定のイベント列を持つものを警告しない機能である．除外機能の実現には，検査仕様に対する除外情報の追記が必要である．しかし，情報の追記に伴ない，検査仕様が複雑になり，パターンでは検査仕様を記述できない可能性がある．パターンを維持したまま複雑な検査を実現するには，単純なモデルを組み合わせることが考えられる．モデルの組み合わせによる検査はモデル間で共有する情報を Mediator を介して通信することで実現できる．テストケースの自動生成をするためには，各モデルから得られるコード片の優先関係が不明確であることから，モデルやアクションに対する優先順序の記述が求められる．

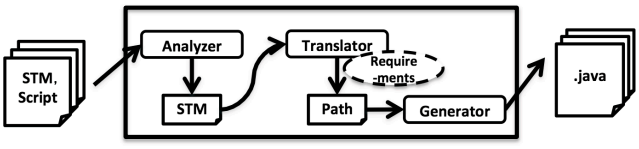


図 5 テストケース自動生成の枠組み

図 5 はこれらの課題を踏まえたテストケースの自動生成手順である．入力された状態遷移モデルとスクリプト記述を Analyzer で解析し，State と Command から構成される STM に変換する．Translator では STM に加え必要に応じてツール利用者の要求を入力する．そして 5 章で示した手順にしたがって，STM とアクション，ツール利用者の要求からテストケースの生成に必要な経路情報を得る．Generator ではここまでで得られた経路情報に対して，各経路に対応するコード片を出力し，最終的なテストケースを得る．

6.2 スクリプトに関する考察

本研究では，FA ではモデル化できない副作用を伴なう検査仕様をスクリプトを用いて FA のアクションとして記述した．過去の JCI の開発実績から検査処理のための再利用可能なコンポーネントは十分に定義されているので，検査処理の大半はコンポーネントの組み合わせのみで記述することができる．したがって，内部構造を知ることなく簡潔な記述で仕様を記述できるので，ツール利用者への負担を最小限に抑えながら検査仕様を可視化することができる．そして，現在の JCI が備える検査仕様は全て状態遷移モデルとスクリプトの組み合わせで表現できることから，検査のカスタマイズのためにスクリプトを導入したことは妥当であると考える．

テストケースの自動生成についても，ノードの情報の取得と比較に関する記述や事後条件に着目することで，木構造の構造や各ノードの情報の詳細化を実現することができた．状態遷移モデルのみではアクション間に共通して扱われる情報に関する操作の記述が困難であったことから，スクリプトの導入は妥当であると考える．

7 おわりに

本研究の成果として，状態遷移モデルを用いた検査仕様記述を可視化した．そして，検査仕様から適切なテストケースの生成し，それをもとにテストをおこなえるようにすることで仕様の実現の正しさを保証できるようになった．今後の課題として，除外機能の追加やメソッド間にまたがる情報を踏まえたテストケースを生成できるようにすることで，より多くの検査の品質保証を実現することが挙げられる．

参考文献

[1] CheckStyle, http://checkstyle.sourceforge.net/.

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

[3] 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.

[4] H. Siy, and L. Votta, ”Does The Modern Code Inspection Have Value?,” Software Maintenance, 2001. Proceedings. IEEE International Conference, 2001, pp. 281-289.

[5] 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・エンジニアリング最前線 2009(情報処理学会 SES2009), 近代科学社, 2009, pp.119-126.