B-016

ソースコードの解析を利用したモデル検査に基づく欠陥抽出手法の提案

Defect extraction technique based on model checking using analysis of source code

青木　善貴†　　松浦　佐江子‡
Aoki Yoshitaka　　Matsuura Saeko

1. はじめに

システム開発は顧客からの要求仕様の正確な理解が必須である．しかし設計者が仕様を正しく理解していても，実際にプログラムを作成するプログラマが仕様を全て正しく理解しているとは限らない．そのような理解に基づいて作成されたプログラムは，一見正しく動作するが不具合を内在していることが多い．

不具合は早期発見が望まれ，発見の方法としては大きく以下の二つがある．

一つはソースコードのレビューである．これは仕様を完全に理解しているものが行えば不具合の発見にかなり有効であるが，ソースコード量が膨大になってくると時間的制約が厳しいため主要処理をチェックするだけで精一杯になり，全ソースコードをチェックすることはきわめて難しく，未チェック部分のミスが見逃されることになる．

もう一つは，テストケースを設定して行う単体テスト，統合テスト等である．これは設定された条件についてはプログラムの正しさを保証するが，テストケースから漏れた条件についてはその限りではない．テストケース作成時にはもちろん網羅的に作成することを心がけているが，仕様を完全に理解している者が全般にわたって作成に携われる訳ではなく，漏れが生じることは間々ある．

したがって，ユーザ引渡し直後の大規模なシステムには何らかの不具合が内在することは避けがたく，これが，思わぬ所で障害を発生させてシステム開発プロジェクトの納期及びコストを圧迫する．開発プロジェクトへの影響を最小限にするためには，不具合箇所の発見及び原因究明を早期に行うことが重要である．

開発中もしくは本番開始後に不具合が発生した場合，開発及び保守担当者は，通常はログ＆エラーメッセージで不具合の場所や原因を特定するが，それができない時は，デバックにより追究を行う．しかし全くの同条件でプログラムを実行することは難しいため再現は困難で，原因の特定に多大な工数を要することが多い．

ソフトウェアの上流工程で信頼性を向上させる技術として注目されているモデル検査は，時相論理式を用いて満たすべき状態を指定することにより，対象システムがその状態に到達するかどうかを網羅的にチェックすることができる．これにより不具合を再現できる可能性が高い．再現しにくい障害の場所や原因を発見するためには有効であるといえる．またモデル検査ツールは指定した内容が成り立たない場合は，反例となる状態を提示するので，シミュレーション機能によりその過程も確認できる．

本研究では，コードレビュー及びテストケースの網羅性不足をモデル検査で補いつつ，低コストで不具合箇所及び原因を発見する手法を構築することを目指す．

本稿では提案した手法を，実際に無限ループを発生させた欠陥を基にして作成した Java サンプルプログラムに適用し，その有効性を確認する．

†芝浦工業大学大学院　電気電子情報工学専攻
Graduate School of Engineering, Shibaura institute of technology Department of electronic engineering and computer science
‡芝浦工業大学　システム理工学部電子情報システム学科
Shibaura institute of technology Department of electronic information system College of Systems Engineering a Science

2. モデル検査に基づく欠陥抽出法

2.1 モデル検査の問題点

本提案では，高い精密性，網羅性で検査ができることを考慮しモデル検査ツールを使用することとした．

しかしモデル検査ツールには以下の点で問題がある．

- モデル検査ツールの記述ルール習熟が必須
  検査モデルはシステム記述言語で記述する必要がある．検査者はこの記述ルールに習熟しなければ検査を行うことができない．
- モデル化範囲の設定の難しさ
  モデル化の範囲を変えたい時の対応の難しさ．

特にモデル化範囲については，不具合の原因がログや，過去の障害記録等より特定できればよいが，そうでないとモデル化の範囲を決定することは困難である．怪しそうな部分をしらみつぶしにモデル化して検査することは可能であるが，作業は見つかるまで何度も行うことになるため，一々手作業でモデルを作成して検査ツールにかけることは現実的ではない．

2.2 提案手法の業務システムに対する親和性

モデル検査は，状態を網羅的に検査するため，調べなければならない状態数が非常に多くなる可能性があり，モデル検査ツールで検査しきれない場合がある．そのため必要最小限の規模で検査をすることが望ましい．

業務システムを構築する場合，実際に業務を行う顧客からは現行の業務プロセスを継続の希望が強く，業務プロセスの変更を伴う業務改善は望まれないことが意外と多い．その場合，業務システムに求められるものは，"正確さ"，"速さ"，"確実さ"，"メンテナンス性の良さ"である．そのためコーディングも安全確実な書き方になる傾向が強く，効率上問題ないことが確認できれば，より高い効率性やコンパクトさを求めて，コーディングを「整理」することは，コードの読み易さを低下させかねないため行われない．

従って，企業における業務システムは，業務全体で見ると複雑であるが，プログラム内の一つ一つの処理については単純なものが多いと言える．処理が単純ならば，特定の処理に着目した構造分析が容易になる．これは，特定の処

理に着目してモデル化を行う今回の手法には都合がよく，最小限の規模での検査が可能になると考える．

2.3 検査プロセス

2.3.1 検査支援ツール

本提案では，低コストで検査を行うことを目的としており，作業負荷を軽減するために検査支援ツールを用意した．ソースコードから不具合現象を発生させている可能性のある箇所の把握を容易にすることと，モデル検査ツールが検査する対象モデルを作成することを支援するものである．

2.3.2 検査手順

本提案手法では，以下の手順で検査を行うことにより，2.1 で述べた問題を回避しつつ，簡単にモデル検査が実行できるようにする．

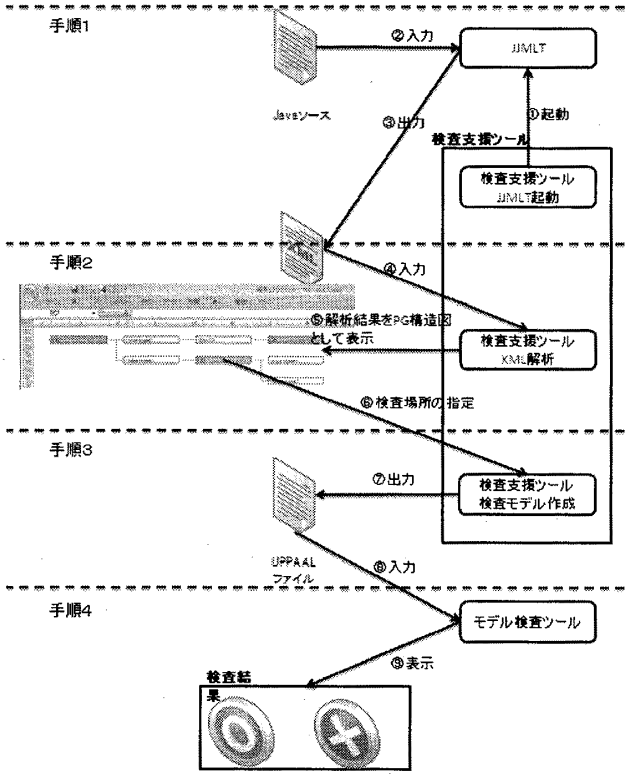


図 1　処理手順概要

手順 1　XML 作成

検査支援ツールより JJMLT[6]を起動して Java のソースを XML 形式のデータに変換する

手順 2　PG 構造図を作成する

推定される障害原因から着目する要素を決定するために，Excel マクロを利用して，プログラムの構成要素に基づきプログラムの PG 構造図を作成する．PG 構造図とは，モデル化の範囲を把握しやすくするため，特定の要素に着目してプログラムの構造を表す，図である．図 2 は"for"と"if"に着目した PG 構造図である．

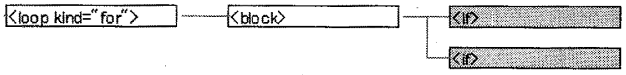


図 2 "for"と"if"に着目した PG 構造図例

手順 3　モデル検査ツール用ファイル作成

構造図より検査部分を選択し，その部分をモデル化してモデル検査ツールが読込める検査モデルを作成する．

手順 4　モデル検査ツールによる検査

モデル検査ツールが作成されたファイルを読込み，検査を行う．

手順 1～3 は検査支援ツールにより自動的に実行する．

2.4 UPPAAL

UPPAAL[7]はスエーデンの UPPSALA 大学とデンマーク AALBORG 大学によって開発されたモデル検査ツールであり，以下の特徴を持っている．

- 時間制約を考慮して検査ができる
- モデルがグラフィカルに作成できる
- 検査結果（反例，シミュレーション）がグラフィカルに確認できる

他のモデル検査ツールとの比較は以下の表のとおりである．

ツール	グラフィカルな結果表示	グラフィカルなモデル作成	システム記述言語への依存
SPIN	○	×	大
SMV	○	×	大
LTSA	○	×	大
UPPAAL	○	○	小

本提案では，検査支援ツールで検査モデルファイルを作成し，それを直接モデル検査ツールに読込ませて検査を行う．そのため検査モデルファイル作成の容易さは重要である．

UPPAAL はシステ記述言語への依存度が低く，モデル図をそのまま表現する形式のファイルを使用しており，検査支援ツールでのファイル作成が容易である．よって今回は，UPPAAL を本提案で使用することとした．

3. UPPAAL モデルへの変換

3.1 変換の概要

本提案では「無限ループ」を発見するために，対象プログラムから特定の構文のソースコードを検査対象として PG 構造図を用いて選択し，UPPAAL のモデルに変換する．以下に，これらのモデルへの変換方法を説明する．

3.2 基本モデル

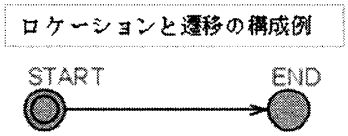
今回の検査対象は" 無限ループ" であるため，それに合ったモデルを想定する．

UPPAAL を構成する基本要素は以下の通りである．

ロケーションは，システムの状態を表し，遷移で結ばれることによりモデルを構成する．

クロック変数を用いた制約を検査モデルに付加できる．

遷移はシステムの振る舞いを表し，遷移する条件設定，変数値の更新，他プロセスとの通信ができる．



モデル化するにあたり注目すべきプログラムの構成要素を絞る．そうすることにより，注目する側面に限定したモデルを定義でき，厳密・網羅的な検査を行うことができる．

今回は[for]と[if]に注目する．（[]は注目するプログラム構成要素を表現するものとする）この二つの要素に注目する理由は，無限ループの多くは，ループ命令の条件式の解釈が不適切なためループから脱出できなくなることが多いことと，特定の条件で発生する（常時発生するならテストで既に発見されている）ことから条件分岐である[if]が関係する可能性が高いということである．この二つの要素を用いてモデルを作成することにより無限ループを検査できると考える．

絞った項目でモデルを構成する基本構造を定義する．

[for]については以下のように定義する．

[for]の式を　for(初期化；　条件(ループ継続)；　更新)と解釈し，for 文の開始時点，条件式の評価開始時点，for 文のブロック終了時点をロケーションとし、"初期化"，"条件(ループ継続)"，"更新"を UPPAAL の遷移に設定する．

また UPPAAL で条件分岐する場合には，明確に条件を指示する必要があるため，条件(ループ継続)の false として条件(ループを抜ける)作成し，ループを抜ける分岐先の遷移に対してこの条件を設定する．

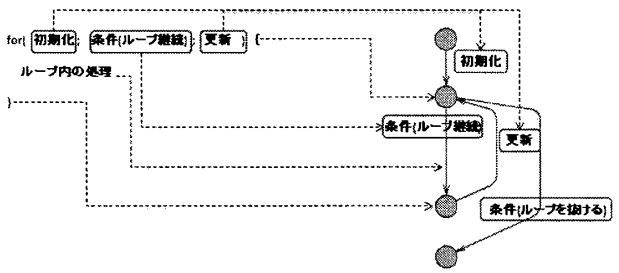


図 3　for 文基本モデル

[if]については以下のように定義する．

[if]の式を　if（条件(true)）　と解釈し条件式評価時点と if 文のブロック終了時点をロケーションとし，"条件(true)"を UPPAAL の遷移に設定する．

また，必ず else の遷移を用意し条件(false)を設定する．

else if で条件を追加する場合は，遷移を追加してそこに条件を設定することにより対応する．

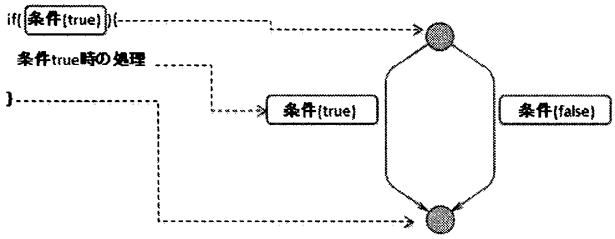


図 4　if 文基本モデル

3.3 変数のモデル化

不具合の原因を特定するためには，検査対象のモデル上でシステムがある特定の状態に到達する場合のソースコード上の変数の値を特定し，それらが外部のどのような入力に起因するかを判定する必要がある．検査対象コードで無限ループを発生させる要因となるのは、ループ条件式や分岐条件式に含まれる変数である．例えば，条件式が比較演算子で定義されている場合には，そのオペランドに相当する式を変数化し，その値を観察することで，無限ループ発生原因を特定する．

基本としてモデル化はその構文内でも最も上位にある変数に合わせて行う．

例えば次のような条件式 "i1 < array.size()"の場合，i1 は int 型の変数，array.size()はメソッドである．この時，i1 はそのまま使用し，array．size()は i1 に合わせ int 型として定義しなおす．検査の時に元のプログラムと対比しやすくするため，定義する変数名は元の名称を生かしたものにする．この例の場合 i1→i1，array.size()→arraySize と定義してモデル化する．

3.4 入力処理及びチェック処理のモデル

与えられているスペックに基づき入力値を発生させるモデルを作成する．プログラムの入力処理部の構造を雛形にしてモデルを作成し，スペックの内容により修正を加える．

3.5 各モデルの接続

UPPAAL では同期チャネル通信を用いることにより他のモデルと通信を行うことができる．この機能を用いて，それぞれ処理毎に作成してきたモデル同士を通信させて対象プログラムの検証対象部分をモデル化する．

4. 適用事例

4.1 適用対象

適用対象プログラムは会計データの検索プログラムである．検索対象になる会計期間の検索範囲(From-To)の二つの値を検索条件として指定すると，その範囲内のデータを処理して請求書を作成する．第 1 会計期間を指定した場合には，前年度の第 13 会計期間を検索範囲として追加して処理する機能があり，この機能が特定の条件で無限ループを発生させる．今回，このプログラムのロジックを Java に置き換え検査対象としてサンプルプログラムを作成した．

```
for(int i=1;i<11;i++){
    System.out.println("会計期間FROM＝");
    String s2 = buf.readLine();
    if(s2.length()>0){
        accPiriod_From = Integer.parseInt(s2);
    }
    System.out.println("会計期間TO＝");
    String s3 = buf.readLine();
    if(s3.length()>0){
        accPiriod_To = Integer.parseInt(s3);
    }
```

図５　適用対象プログラム(入力部)

```
import java.util.ArrayList;

public class TableAdd {
    public void  table_add(ArrayList<int[][]> array ){
        int[][] accPiriod;

        for(int i1 = 0 ;i1 < array.size();i1++){
            accPiriod = (int[][]) array.get(i1);
            if(accPiriod [0][0]==1 || accPiriod [0][1]==1){
                array.add(new int[][]{{13,accPiriod [0][1]}});
            }
        }

    }
}
```

図６　適用対象プログラム(会計期間追加)

4.2 適用モデル

個別に作成した検査対象モデルと対象プログラムの入力処理モデル，入力値チェックモデルを UPPAAL の同期チャンネル通信により連携可能なかたちに接続する．さらにモデル検査用に入力値発生モデルと無限ループ状態モデルも同様に接続し適用モデルを構築する．

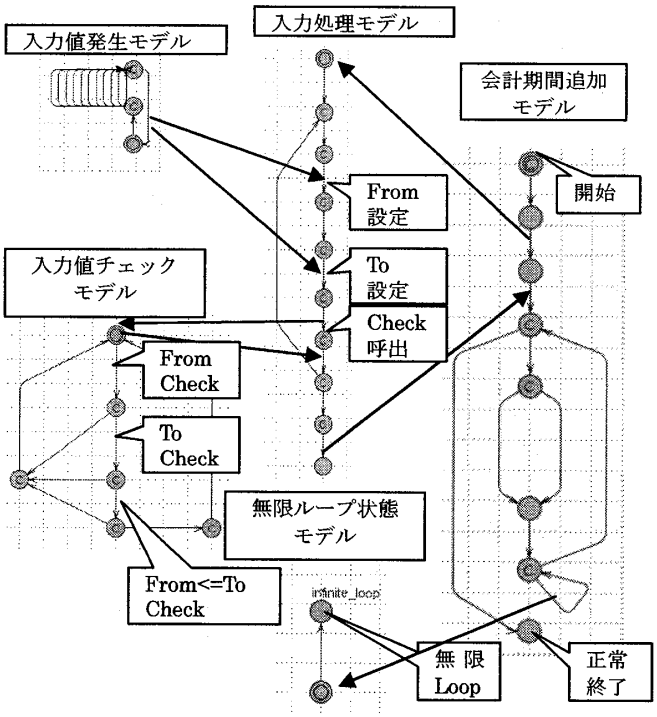


図７　適用モデル

- 入力値発生モデル：1～12 の値をランダムに発生させる．
- 入力処理モデル：検索範囲 From と検索範囲 To の二つを設定する．
- 入力値チェックモデル：検索範囲である会計期間が 1～12 であることと，検索範囲 From<=検索範囲 To であることを保障する．
- 会計期間追加モデル：検索範囲 From=1 もしくは検索範囲 To=1 ならば検索範囲 From=13 の検索条件を追加する．
- 無限ループ状態モデル：UPPAAL は時間制約を考慮した検査ができるのでこれを利用し，一定時間ループを繰り返したら無限ループ状態に遷移する．

4.3 適用結果

UPPAAL を用いて検査条件 A[] not Process4.infinite_loop (いかなる場合も Process4 の infinite_loop の location には達しない)を実行して検査を行う．結果，検査条件が満たされないことが確認できた．

A[] not Process4.infinite_loop
属性は満たされませんでした

反例を再現し，動きおよび発生条件を確認すると図８の結果が得られた．

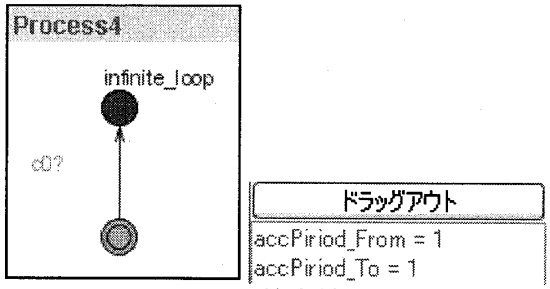


図８　検査結果

UPPAAL は無限ループの状態に遷移し終了した．この時に得られた無限ループになる検索範囲(From-To)の値は，実際のプログラムでデバックして確認した内容と合致した．

4.4 検査対象の絞り込み

今回は，単純なサンプルであったため特にどの部分をモデル化するかでは悩む必要はなかったが，通常の開発現場で使用する場合は，場所を絞り込んでいく作業が必要になってくる．

PG 構造図により検査対象となる構文間の構造は明確であり，検査対象として選択した先頭項目以下は全て検査対象になるので，始めは大きな単位で検査して徐々に範囲を絞っていく手順が可能である．

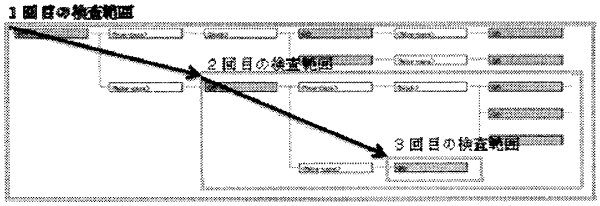


図 9 検査範囲の絞り込みのイメージ

また，小さな単位の検査を繰り返す方法も可能である．

5. 関連研究

関連研究としては，経験の浅い開発者でも検証ができる手法[1]や最小限の規模で検査を行う手法[2]，モデル検査のガイドライン[3]など，本研究でも指摘しているモデル検査ツールの問題点について対応しているものがある．これらは手法を人手で踏襲するものであり，本提案手法にある自動化の観点は無い．

また，本研究でも目指している自動的な検査モデル作成については，UML 図より自動的に検査モデルを作成する手法[4][5]がある．これらは，UML 図の使用を前提としていることからもわかるように上流工程での使用を想定している．それらと違い本提案手法は，ソースコードを基に必要最小限の規模でモデルを自動生成し，検査範囲を絞り込みながら検査を行うもので，実際にプログラムが使用される現場サイドでの使用を強く意識したものとなっている．

6. まとめと今後の課題

本研究では，着目すべき要素を決めてモデル検査を行うことにより，低コストに不具合箇所や原因を発見する手法を提案した．今回，無限ループの不具合を例にとり，本提案手法を適用し原因が検出できたことにより，本提案手法の有効性が確認できた．

今後は，より適用できる要素を増やすことにより他の不具合事例についても確認していきたい．課題としては，検査部分における変数と外部から与えられる入力値の関係を明確にすることによりモデル検査の入力値範囲の精度を上げることである．

参考文献

[1]長野伸一, 吉岡信和, 田原康之, 本位田 真一, ソフトウェア設計に対するモデル駆動型検証プロセス(ソフトウェア分析・設計技法), 情報処理学会論文誌, VOL.47,NO.1PP193-208(2006)

[2] 岡野 浩三, 楠本 真二, UML/OCL に記述された時間 QOS の階層的検証手法の提案, 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, VOL106,NO/202PP13-18 (2006)

[3]青木翼, 長谷川哲, 夫宮本博, 暢渡邊竜明, UPPAAL によるモデル検査適用ガイドラインの作成(解析・検証),情報処理学会. ソフトウェア工学研究会報告, VOL. 2008,NO.29PP203-210(2008)

[4] 長谷川哲夫, 深澤良彰, シーケンス図からの時間性能モデル検査用オブザーバ生成手法(テスト・検証), 情報処理学会. ソフトウェア工学研究会報告, VOL.2006,NO.35PP143-150(2006)

[5] 田昭彦, 深海悟, 形式的検証用モデル自動生成機能を持つ上流工程支援システムの開発(形式手法(2)), 情報処理学会. ソフトウェア工学研究会報告, VOL.2008,NO.29PP195-201(2008)

[6]JJMLT:HTTP://WWW.HPC.CS.EHIME-U.AC.JP/~AMAN/PROJECT/JJMLT

[7]UPPAAL:HTTP://WWW.UPPAAL.COM/

[8]田中譲,ソフトウェア科学基礎,近代科学社,2008