モデル検査による仕様とソースコードの不一致発見

Approach for Detection of Inconsistency between the Specification and Source Code by using Model Checking

青　木　善　貴

要　約　システム開発では不十分な要件定義や実装時のミス・誤解などにより人手では原因の特定が困難な不具合が発生する．不具合の解消にはまず再現確認を行うことが前提となるため，再現確認ができないと不具合の原因の特定が困難になる場合が多い．これは目に見える不具合の現象と原因が乖離していて直感的に原因を想定することが難しいために発生する．こうした不具合は成果物の仕様についての妥当性・整合性を漏れなく検査することにより発見できると考えられる．

システムが取りうる様々な状態を検証するモデル検査が近年注目されている．モデル検査はシステムの振舞いがある性質を満たすか否かを網羅的な探索により自動的に検証する方法であるが，適用にはモデル検査の技術・知識を習得する必要があるためシステム開発への適用は進んでいない．本稿では業務ドメインの知識や開発業務で一般的に使用される技術を利用してモデル検査を行う手法を提案する．提案手法は，成果物の品質の向上や再現困難な不具合の原因の特定に効果が見込める．

Abstract　Software programs often include many defects that are not easy to detect because of the developers' mistakes, misunderstandings caused by the inadequate definition of requirements, and the complexity of the implementation. Premise that the problem (defect or bug) must be reproducible so as to identify the cause and to solve the problem. Deviation between a phenomenon of failures and the cause is one of the main factors which make it difficult to detect the defects. When a Specification of a document is inspected about its appropriateness and consistency without omission, these failures can be discovered.

Model checking has been favored as a technique to improve the reliability earlier in the software development process. We have proposed the source code verification method to find the discrepancy between the behavior of the source code and the specifications by using model checking technology.

Model checking can automatically verify system behavior by exhaustive checking. However, model checking is not widespread because you must learn about the knowledge of model checking.

By this proposed method, we can expect for higher quality artifacts and for the effect of detecting defects that are difficult to reproduce.

1. は　じ　め　に

システム開発では不十分な要件定義やシステムの実装時のミス・誤解などにより大小多くの不具合が発生する．不具合の解消にはまず原因を特定して再現確認を行うことが前提であるが，目視できる不具合とその原因が乖離していて直感的に原因を想定し難い場合は，不具合の原因の特定が困難になる．

このような不具合の原因には，想定外の業務ルールの組み合わせや，意図しないタイミング

での複数機能の同時実行など，要求仕様と実際に作成されたプログラムの間での，システムの振舞いの不一致が挙げられる．本稿では，システムの振舞いとはシステムにより自動化した処理とユーザの手作業の組み合わせで表される動作の連なりとする．

仕様書には，システムで実現したいシステムの振舞いが記述されている．このシステムの振舞いを理解して設計し，実装したものがシステムである．仕様に記述されている振舞いが，システム上で実現できない，もしくは仕様に記述されていない振舞いがシステム上で実現できてしまうことを本稿ではシステムの振舞いの不一致と呼ぶ．

このような不一致の発見は，通常は単体・結合テストにより行うが，システムの振舞いの不一致を漏れなく抽出するテストケースの作成は困難である．そのため，システムの振舞いの整合性を漏れなく検査する仕組みが必要である．その仕組みのひとつとして「モデル検査」が考えられる．

モデル検査は，システムを有限の状態遷移で表したモデルに対し，要求する性質を時相論理式で記述し，モデルがこの論理式を満たせるかを検証する手法であり，システムの振舞いを漏れなく検査することに適している．モデル検査はシステム開発に有効な手法であることは間違いないが，一般的なシステム開発への適用は進んでいない，それはモデル検査用のモデル作成や検証式の作成にモデル検査特有の知識・技術が必要だからである．本稿では，モデル検査のシステム開発への適用を容易にするために，仕様とソースコードの不一致を発見する「ソースコード検証手法」を提案する．

従来からあるモデル検査の適用事例は，仕様もしくはソースコードを個別にモデル化して検証することが多い．また，モデル検査の多くの研究はモデルの抽象化やモデル検証の効率化など，モデル検査の技術的な面の改善に注力しており，実際のシステム開発においてモデル検査の普及を進める研究にはあまり注力していない．本研究はそうした従来の研究とは異なり，モデル検査のシステム開発への適用を進める点に注力している．これはシステム開発において大きなメリットがあると考えられる．筆者はモデル検査を用いてソースコードから不具合を発見する研究[1][2]を行ってきており，本稿はそれを基に執筆している．2 章でモデル検査の利点と問題点，3 章で既存研究を説明し，4 章で本研究の提案手法を解説する．

2. システム開発におけるモデル検査適用の利点と問題

2.1 モデル検査の概要

形式手法は，数学的に厳密に意味付けられた言語を用いて情報システムの要求や設計などを記述し，情報システムがユーザの要求を満たしているかを論理的に検証する手法である．形式手法はシステムが持つ性質そのものを検証できるため，論理的に不具合が存在しないことを保証できる．そのため欧米を中心に実際のシステム開発への適用事例は増加している[3]．

形式手法は，二つのアプローチに分けられる．ひとつはモデル検査であり，システムの振舞いを状態遷移系のモデルに変換し，それが満たすべき性質を時相論理式*1 で表して充足関係を検証する．検証は全自動で網羅的に行われる．性質が満たされない場合は，満たされない状態に至る状態遷移の実例である反例を出力する．また対象の状態は有限である必要がある．代表的なモデル検査ツールは SPIN[4]，NuSMV[5]，Java Pathfinder（JPF）[6]，UPPAAL[7]がある．

もうひとつは，数学的基盤に基づく形式仕様記述言語で記述した仕様を「定理」とみなし，Hoare 論理*2 などのプログラム検証の論理体系を「公理」として，検証したい性質が満たされる

ことを推論規則に基づいて証明する演繹手法である．代表的な手法にはVDM[8]やB-method[9]などがある．対象の状態が有限である必要はない．検証には人による作業が必要であり，全自動での検証はできない．

本研究はシステム開発への適用を目的としているため，数学的知識がなくても形式手法を利用して全自動で検証ができるモデル検査を使用する．

2.2　システム開発におけるモデル検査適用の利点

想定されるエラーのテストで期待される結果を，エラー処理の正常終了と考えると，単体テストや結合テストで期待される結果は，ほぼ正常終了といえる．異常終了を発生させるテストケースを作成することは難しく，またユーザが実施する業務を異常終了もすべて網羅してテストすることは困難である．

モデル検査は適切な検査モデルを作成することで，システムの状態をすべて検証することができる．つまり，通常のテストでは見逃されがちな不具合の発見も可能であり，未知の不具合を探す場合や，単体テストや結合テストの補完に利用できると考えられる．モデル検査を行って不具合が発見されなければ，論理的に不具合が存在しない保証になる．

不具合対応で，既知の不具合の原因を特定する場合，モデル検査の網羅的な検証は，不具合の原因の特定に有効である．

2.3　システム開発におけるモデル検査適用の問題

モデル検査を適用するには，特有の知識・技術の習得が必要であり，これがシステム開発への適用が進まない理由の一つである．

例えば，ソースコードについてモデル検査を行う場合，検証したい性質をソースコード上のステートメントと関連付けて検証式を記述する必要がある．そのためには，ソースコードとシステムの振舞いを変換した検査モデルを読み解く知識と検証式に変換する技術が要求される．また，検証結果から不具合の原因を特定する場合，モデル検査が出力する反例を解析するには状態遷移の各状態を検査モデルと照らし合わせて読み解く知識が要求される．

モデル検査はシステム開発で有効な検査手法であるが，その適用を促進するためにはモデル検査の知識・技術面の難易度を下げる必要がある．

3.　モデル検査の研究動向

本研究の特徴に関わるモデル検査の研究動向と，本研究の位置づけについて述べる．

ソースコードの不具合検出にモデル検査を使用する研究は多く，それらは「ソースコードの制御構造の検査」と「仕様に関わる性質の検査」に分類できる．「ソースコードの制御構造の検査」ではデッドロックやデッドコードの発見の研究が盛んである[10][11]．「仕様に関わる性質の検査」は，ソースコードと仕様を共に理解して検証式を作成して検証するものである．ソースコードにアサーションを埋め込みそこへの到達可能性により検証する研究[12][13][14]と，検証を容易にするために，特定の機能，例えばファイルI/Oの機能をあらかじめモデル化して検証する研究[15][16]がある．これらの研究の検証では複雑な検証式が用いられている．また特定の機能のモデル化は検証対象がその特定の機能の振舞いに限定される．本研究は「仕様に関わる性質の検査」の分類に属する．本研究は仕様とソースコード間の不一致を発見すればよいため検

証式は複雑にならず，検証対象も限定されず幅広い適用が可能である．

検証式をいかに正しく，容易に作成するかという観点の研究も行われている．Dwyer ら[17]は検証式の作成を容易にするために仕様を分類してパターン化したプロパティ仕様パターンを作成し，これを検証式と結びつけて，パターンが定まれば検証式を決定できるようにしている．Rachel ら[18]はプロパティ仕様パターンの決定をツール化した．これらの手法は検証式の記述の補助には確かに有効だが，使用にはプロパティ仕様パターンの意味を正確に理解する必要があり，難易度が高い．Dwyer の研究[17]では 555 個の仕様を集め彼らが提案するプロパティ仕様パターンにマッチするか確認している．結果として，555 個のサンプルのうち 511 個（92%）がマッチすることが確認されている．本研究は，開発業務で一般的に使用されるデシジョンテーブルを用いたシステムの振舞いの把握により，容易に検査式を作成できる．また本研究の適用範囲は主なプロパティ仕様パターンに該当するため同様の検証が可能であり，511 個のサンプルの約 80%を網羅できると考えられ，有効性は充分にあると考えられる．

反例は不具合の原因の特定において重要な情報であるが，特定作業は仕様，検査モデルを組み合わせて解析する必要があるため難しい．反例解析の研究は「反例を理解しやすくする」と「反例を収束させる」に分類できる．「反例を理解しやすくする」ではモデルをコンパクト化する Chan らの研究[19]と複雑な状態遷移を構造化して見やすくする Clarke らの研究[20]がある．「反例を収束させる」は，検証目的と異なる反例を淘汰して目的と合致する反例に収束させることを目的とした研究である．実際の環境では成立しない偽反例を見つける Cong らの研究[21]や，シミュレーションの技術を用いて反例を検索する Lerda らの研究[22]がある．本研究は上記の両方の分類に属する．本研究は「反例を理解しやすくする」については，反例をグラフ化し視覚的に理解を補助する．また，「反例を収束させる」については，検証式を修正して再帰的に検証することで問題となる状態遷移の特定を行う．

4.　ソースコード検証手法

4. 1　提案手法の概要

本稿では，モデル検査のシステム開発への適用を促進するためにソースコード検証手法を提案する．ソースコード検証手法は，ユーザがシステムの仕様に基づき作成した検査モデル（以降，仕様モデル）とソースコードから制御フローに基づき作成した検査モデル（以降，ソースコードモデル）を結合させて，特定の性質について検証することで，仕様とソースコード間の不一致を発見する手法である．仕様とソースコードの不一致は，仕様モデルとソースコードモデルの状態遷移の不一致として発見できる．

3 章で述べたとおり，本研究は仕様モデルとソースコードモデルの不一致を発見すればよいため複雑な検証式を必要としない．また業務ドメインの知識や開発業務で一般的に使用される技術による検証式の作成が可能でかつ適用範囲が広い．反例解析の容易化の取り組みも行っている．さらに検査モデルの作成を，開発した検査支援ツールで支援することによりモデル検査の難易度を下げることを図っている．

提案手法では，ユースケースで定義されるレベルのシステムの振舞いを検査の対象としている．ユースケースを構成する要素はアクションと基本フロー，事前・事後・分岐条件である．これらはシステムの基本機能を表しており，仕様書内の記述と対応付けが可能である．したがって仕様モデルとソースコードモデルも同様に対応付けが可能であり，検査モデルが構築で

きる．

図１は提案手法の概要図である．検査の手順を以下の(1)〜(8)に示す．

(1) システムの振舞いは仕様書に記述されているため，そこからアクションと条件（事前・事後・分岐条件），実行結果（状態）を抽出する（図１①）．

(2) (1)の内容をデシジョンテーブルで整理する（図１②）．自然言語で記述された仕様を整理するためにデシジョンテーブルに実行した「結果」の項目を追加する．

(3) 整理結果でシステムの振舞いを状態遷移に変換して仕様モデルを作成する（図１③）．「結果」を状態に置き換え，それらの状態遷移をデシジョンテーブルの「アクション」により起こるものとして作成する．

(4) ユースケースレベルの機能（アクション）はメソッドに対応すると仮定できるため，メソッド単位で機能を捉え，ソースコードの階層構造に基づいたモデル変換を検査支援ツールにより行い，ソースコードモデルを作成する（図１④）．

(5) メソッドをソースコードと仕様の接点として捉えて，ソースコードモデルから仕様モデルへの状態遷移の指示を中継するアクションモデルを定義する（図１⑤）．

(6) ソースコードモデルのメソッド呼び出し部と仕様モデルの状態変化を表す部分をアクションモデルで紐付けして検査モデルを構成する（図１⑥）．

(7) 仕様モデルが想定外の状態にならないことを検証する検証式を作成する（図１⑦）．

(8) この検査モデルを検証式により検証する（図１⑧）．

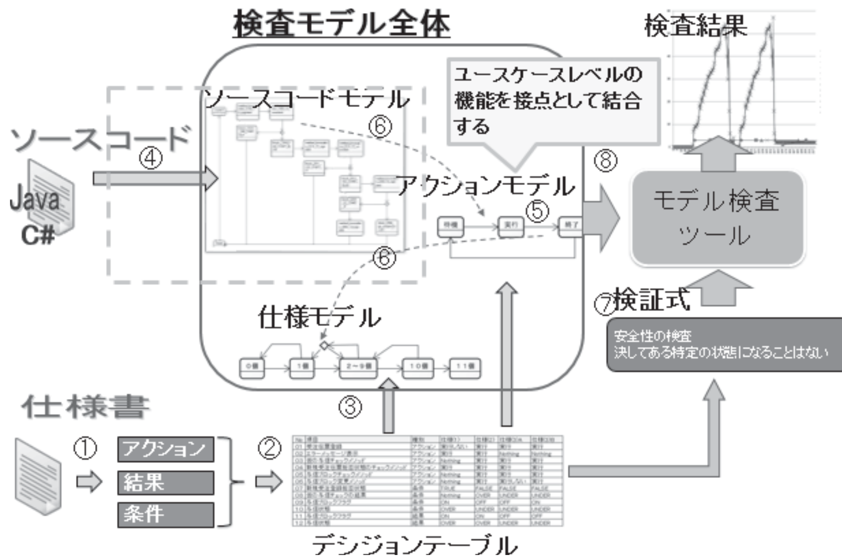

図１　提案手法概要図

(6)で構成する検査モデルの具体例を図２に示す．この例は，長方形を線描するプログラムにおける長方形を追加する処理についての検査モデルである．長方形は10個以下という仕様である．ソースコードモデルの長方形を追加するメソッド呼び出し部と仕様モデルの長方形が増える状態変化を表す部分をアクションモデルの「実行」で紐付けし，検査モデルを構成する．

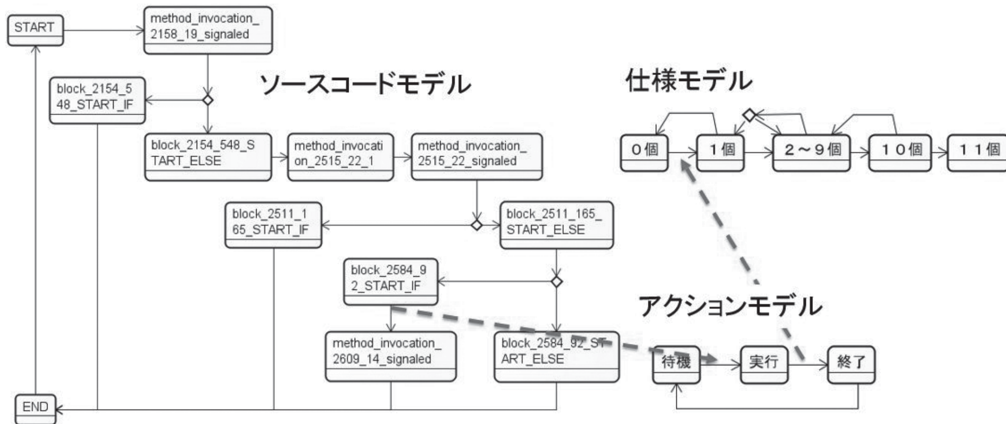

図2　長方形を追加するプログラムの検査モデル

(1)～(3)は基本的に手動で作成する部分であり，定式化する方法を検討中である．(4)は検査支援ツールにより自動で作成する部分である．自動作成で対応できない部分は手動で対応する．(5)～(8)は手動作成だが定式化しつつある部分である．手動の部分は処理内容についての判断が必要になるため定式化が難しい．特に(1)～(3)の部分については仕様書の記述内容によって作業量が大きく変わる．

本研究はユースケースレベルのシステムの振舞いの仕様をデシジョンテーブルで整理してモデル化し，想定外の振舞いがないことを検証している．したがってハードウェア・開発環境・パッケージソフトなどに関する不具合は本研究の提案手法の対象外である．

4. 2　検査実施の詳細

検査の実施は以下の手順で行う．図1⑥で構成した検査モデルはユースケースレベルの機能を表したものになる．

手順1：この検査モデルがソースコードの制御フローの動きを再現できていることを到達可能性の検証で確認する．

手順2：作成した検証式による検証でシステムの振舞いに合致しない状態遷移が起きないことを確認する．

手順3：アクションモデルの基になったメソッドのロジックに問題がある場合，これまでの検証では不具合の原因の特定はできない．その場合は，アクションモデルをソースコードモデルへ変換し，新たなソースコードモデルを追加した検査モデルで手順1～2を再度行う．これによりモデルが詳細化され，該当メソッドの検証ができるようになる．この検査の手順を不具合の原因が特定できるまで繰り返す．ただし，Java クラスライブラリのメソッドなど，ソースコードモデルへの変換ができない要素がモデル化の対象になるまで詳細化しても原因が特定できない場合は，ソースコード以外の要素に問題があることになる．

4. 3　反例の可視化と精度向上

反例に関する取り組みは二つある．一つ目は反例を直感的に理解できるようにグラフ化する

ことである．反例から任意の状態を抽出してグラフ化することにより，単なる状態遷移の視覚化でなく特徴的な変化が俯瞰できるため，反例を理解しやすくできる．グラフ化は本研究で開発したグラフ化ツールを用いて行う．

二つ目は反例の結果の精度の向上である．出力された反例を基に状態遷移を限定する式を作成し，検証式に付加して再検証するというサイクルを繰り返すことにより反例の精度を上げて，不具合箇所を特定しやすくする．

4.4　適用事例

本研究の適用範囲は，3章で述べたプロパティ仕様パターン[17]の主な適用範囲に該当することから，多くのシステム開発に適用できると考えられる．これまでに，実際に提案手法を適用して不具合を発見した事例について述べる．

芝浦工業大学のプログラミング演習の授業で学生が作成したソースコードに適用して，挿入ミスや条件式の記述ミスを発見した．さらに，社内業務システムへの部分的な適用を実施し，不具合になりうる部分の発見もできた．また芝浦工業大学が採択された2014年度IPA「ソフトウェア工学における先導的研究支援事業」においても，授業支援システムを対象とした検証に提案手法を使用して，ステータス変更ロジックのミスを発見している．

5.　お　わ　り　に

本稿ではモデル検査を容易にシステム開発へ適用するための，ソースコード検証手法を提案した．

ソースコードモデルの作成は，直にモデルをさわらなくても作成できる検査支援ツールの導入により対応した．これにより制御フローに基づいた検査モデルの自動作成が可能になった．ただし現状は手動により検査モデルを修正する必要がある．検証式の作成については，仕様を分析して，拡張したデシジョンテーブルに記述しシステムの振舞いを捉えることにより対応した．検証式はこのシステムの振舞いを変換すれば作成できる．モデル検査適用の難易度は軽減されたと考えられる．ただし，現状はデシジョンテーブルの項目の検査モデルの項目への置き換えは手動で行っている．自動化については今後，検討する．

反例の解析は，状態遷移の特徴的な部分を指定してグラフ化することにより，着目する点が明確になり反例の意味が分かりやすくなった．まだ表示方法については充分ではない部分があるため，直感的に理解できる表示についてさらに検討する．

現状では対象のシステムに対して仕様をどこまでどのようにモデル化すれば何を保証することができるのかが明確に定義できていないため，適用判断が難しいという課題がある．したがって今後は，対象のシステムの仕様をどこまでどのようにモデル化すれば，何を保証することができるかを示す保証マップとモデル検査技術導入のガイドラインを事例ベースで作成してシステム開発への適用を図る．

＊1　時間の経過で変化する性質を表わす式．時相論理式にはLTL（Linear Temporal Logic）とCTL（Computation Tree Logic）の2種類がありどちらを使うかはモデル検査ツールにより異なる．

＊2　アントニー・ホーアが発表したプログラムの正当性を論じるための論理的規則群．

参考文献

［1］ 青木善貴，松浦佐江子，モデル検査を利用した仕様とソースコード間におけるシステムの振舞いの不一致発見，ソフトウェアエンジニアリングシンポジウム2014論文集，2014，pp.212-213.

［2］ 松浦佐江子，小形真平，青木善貴，矢沢智史，西村一彦，要件定義プロセスと保守プロセスにおけるモデル検査技術の開発現場への適用，Information-technology Promotion Agency (IPA) SEC journal37 JUL, 2014，pp.8-15.

［3］ 形式手法の実践ポータル，http://formal.mri.co.jp/db/fmtool/，2015.

［4］ SPIN，http://spinroot.com/spin/whatispin.html，2015.

［5］ NuSMV，http://nusmv.fbk.eu/，2015.

［6］ Pathfinder，http://javapathfinder.sourceforge.net，2015.

［7］ UPPAAL，http://www.uppaal.com/，2015.

［8］ VDMTools，http://www.vdmtools.jp/，2015.

［9］ K. Lano;H. Haughton, “Specification in B: An Introduction Using the B Toolkit”, Imperial College Press, 1996.

［10］ Ngui, J.; Strooper, P.; Wildman, L.; Wojcicki, M., “Comparing the Cost-Effectiveness of Statically Analysing and Model Checking Concurrent Java Components for Deadlocks”, Software Engineering Conference, 2007. ASWEC 2007. 18th Australian, April 2007, pp.223-232, 10-13.

［11］ Hao Zheng; Yingying Zhang, “Local State Space Analysis Leads to Better Partial Order Reduction”, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol.33, no.6, June 2014, pp.839-852.

［12］ Beyer, D.; Henzinger, T. A.; Jhala, R.; Majumdar, R., “An Eclipse Plug-in for Model Checking”, Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on, Digital Object Identifier: 10.1109/WPC.2004.1311069, Publication Year: 2004, pp.251–255.

［13］ Lei Wang; Qiang Zhang; Pengchao Zhao, “Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking”, Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on, Digital Object Identifier: 10.1109/SCAM.2008.24, Publication Year: 2008, pp.165-173.

［14］ Thompson, S.; Brat, G., “Verification of C++ Flight Software with the MCP Model Checker”, Aerospace Conference, 2008 IEEE, March 2008, pp.1-9, 1-8.

［15］ Xiaoli Gong; Jie Ma; Qingcheng Li; Jin Zhang, “Automatic Model Building and Verification of Embedded Software with UPPAAL”, Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on, Nov. 2011, pp.1118-1124, 16-18.

［16］ M. Achenbach; K. Ostermann, “Engineering Abstractions in Model Checking and Testing”, Source Code Analysis and Manipulation, Proc. of SCAM ’09., 2009, pp.137-146.

［17］ M. B. Dwyer; G. S. Avrunin; J. C. Corbett, “Patterns in property specifications for finite-state verification”, Proceedings of the 1999 International Conference on Software Engineering (ICSE), 1999, pp.411-420.

［18］ Rachel L. Smith; George S. Avrunin; Lori A. Clarke; and Leon J. Osterweil, “PROPEL: an approach supporting property elucidation”, Proceedings of the 24th International Conference on Software Engineering (ICSE ’02), 2002, pp.11-21.

［19］ Chan, W.; Anderson, R. J.; Beame, P.; Burns, S.; Modugno, F.; Notkin, D.; Reese, J. D., “Model checking large software specifications”, Software Engineering, IEEE Transactions on, vol.24, no.7, Jul 1998, pp.498, 520.

［20］ Clarke, E.; Jha, S.; Yuan Lu; Veith, H., “Tree-like counterexamples in model checking”, Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, 2002, pp.19-29.

［21］ Cong Tian, Zhenhua Duan, “Detecting Spurious Counterexamples Efficiently in Abstract Model Checking”, Software Engineering (ICSE), 2013 35th International Conference on, 2013, pp.202-211.

［22］ Lerda, F.; Kapinski, J.; Maka, H.; Clarke, E. M.; Krogh, B. H., “Model checking in-the-loop: Finding counterexamples by systematic simulation”, American Control Conference, 2008, pp.2734-2740.

（参考文献中のURLは，2015年8月7日時点での存在を確認.）

執筆者紹介　青　木　善　貴 (Yoshitaka Aoki)

1991 年日本ユニシス(株)入社．製造流通部門でシステム開発プロジェクトに従事．2011 年より総合技術研究所にて研究活動を行っている．芝浦工業大学 SIT 総研 客員研究員．知能ソフトウェア工学研究会専門委員．