有界モデル検査を用いた逐次 C プログラムの
モジュラー検証に関する研究

橋本 祐介

博士 (情報学)

総合研究大学院大学
複合科学研究科
情報学専攻

平成２３年度
(2011)

2012 年 3 月

本論文は総合研究大学院大学複合科学研究科情報学専攻に
博士（情報学）授与の要件として提出した博士論文である。

審査委員：

中島 震（主査）	国立情報学研究所／総合研究大学院大学
岡野 浩三	大阪大学
日高 宗一郎	国立情報学研究所／総合研究大学院大学
細部 博史	国立情報学研究所／総合研究大学院大学
劉 少英	法政大学

（主査以外はアルファベット順）

Modular Verification of Sequential C Programs using Bounded Model Checking

Yuusuke Hashimoto

DOCTOR OF
PHILOSOPHY

Department of Informatics
School of Multidisciplinary Sciences
The Graduate University for Advanced Studies (SOKENDAI)

March 2012

A dissertation submitted to
the Department of Informatics,
School of Multidisciplinary Sciences,
The Graduate University for Advanced Studies (SOKENDAI)
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

	Advisory Committee:
Shin Nakajima (Chair)	National Institute of Informatics / The Graduate University for Advanced Studies
Kozo Okano	Osaka University
Soichiro Hidaka	National Institute of Informatics / The Graduate University for Advanced Studies
Hiroshi Hosobe	National Institute of Informatics / The Graduate University for Advanced Studies
Shaoying Liu	Hosei University

(Alphabet order of last name except chair)

内容梗概

近年，身の回りの消費者向け製品や社会基盤システムでは，多くの機能がソフトウェアで実現されている．製品やシステムの障害が社会的関心事となり，ソフトウェアにも高い品質が求められている．産業界ではソフトウェアの品質確保をプログラム・テスティングで行っており，開発期間の半分以上を費やすこともある．ソフトウェア品質向上の新しい技術が求められている．

自動検証の技術にロジック・モデル検査の方法がある．有限遷移状態系として表現された検証対象と時相論理で表した検証性質を与えると，有限状態空間上の実行可能な全経路を網羅探索して，検証性質が満たされることを調べる．モデル検査には,「状態爆発」と呼ばれる，検証対象の規模に関して指数的に計算資源を消費するスケーラビリティの問題がある．有界モデル検査（BMC）は，探索範囲を限定して状態爆発を緩和し，効率的に不具合を発見する手法である．プログラムを対象とするソフトウェアモデル検査は，プログラムを有限状態遷移系に変換してモデル検査を行う．自動検証かつ網羅性が高いことから，産業界に導入しやすい品質向上手段といえる．しかし，実用的なプログラムでは BMC でも状態爆発が起こる．従来のプログラム・テスティングとの関係も明らかでない．ソフトウェアモデル検査の実用化では，これらの問題を解決する必要がある．

プログラム検証の考え方にモジュラー検証がある．C プログラムの場合，関数に事前・事後条件を，大域変数に不変条件を付記し，事前・不変条件を満たす状態から関数を実行し，実行後の状態が事後・不変条件を満たすことを静的に調べる．個々の関数を検証単位とするので，大規模なプログラムも扱える．しかし，関数に閉じて検証するので，コールシーケンスに関わる情報の不足から，不具合の見逃しや誤警告が起こりうる．

本研究では，逐次 C プログラムの新しい品質向上手段として，BMC とモジュラー検証を組み合わせたソフトウェアモデル検査技術を提案する．モジュラー検証の導入により BMC のスケーラビリティを向上する．この BMC を用いたモジュラー検証を本研究の基本的な技術として，さらにモジュラー検証の課題と BMC の課題についてそれぞれ解決方法を提案し，その有効性を実験により確認する．

BMC を用いたモジュラー検証では，事前・事後・不変条件を，assert や assume といった BMC の検証プリミティブに変換して対象プログラムに埋め込む．そして，事前・不変条件を満たす初期状態から到達可能な状態が事後・不変条件を満たすことを，BMC の方法で調べる．産業界の製品プログラムを用いた実験を行い，別途実施した単体テストで見つかった不具合を再発見することで提案方法の基本的な有効性を確認した．

C プログラムでは，モジュールへの再入や関数ポインタを用いた間接呼び出しといった従来の静的なモジュラー検証では不具合を見逃す状況が発生する．

特定の機能を実現する関数の集まりをモジュールと呼ぶとき，あるモジュールから呼ばれたモジュールが，呼び出し元モジュールを呼び返すことを，モジュール再入と呼ぶ．モジュール再入はコールバック型プログラムでは頻繁に起こる．再入箇所で不変条件違反が起きる場合に，関数を検証単位とするモジュラー検証では不具合を見逃す．本研究では，ファイルをモジュールとみなし，あるファイルに定義された不変条件をそのファイルに限定して検証するための記法と，ファイル限定の不変条件から検証プリミティブへの変換方法を考案した．これを用いて，検証単位をコールシーケンスに拡張する BMC のインライン展開機能と組み合わせた検証方法を提案する．提案方法の実験を行い，モジュール再入箇所において，ファイル限定不変条件への違反を精度良く発見できることを示した．

関数ポインタを用いた間接呼出しでは，実際に呼ばれる関数は，そのアドレスを関数ポインタに実行時に代入することで決まるので，静的に特定することが難しい．また，間接呼び出し側と実際に呼ばれる側が独立に開発されることもあり，前者の検査を行う際に後者のコードや事前・事後条件を利用できないことも多い．本研究では，関数ポインタ変数に事前・事後条件を定義する記法を導入したモジュラー検証と，関数ポインタと実際に呼ばれる関数の置換可能性検査との 2 段階からなる検証方法を提案する．置換可能性検査には，オブジェクト指向プログラミングにおけるスーパー・サブタイプの置換可能性の考え方を応用した．オープンソースの OS である MINIX を用いて提案方法の実験を行い，関数ポインタの事前・事後条件によって誤警告を減少できることを確認し，間接呼び出しに関わる未発見バグの検出に成功した．

BMC は，プログラムを有限状態遷移系に変換する際に，抽象化を行わず，精度の高い検証を行う．しかし，変換時の過小近似によって，有限状態空間上の経路が少なくなり，元のプログラムに存在する不具合を見逃すことがある．そこで探索しない過小近似箇所を見つけて，別の手段で検査する必要がある．本研究では，BMC の過小近似箇所を自動検知する方法を考案し，検知した場合に，事前・事後条件からテストケースを自動生成して，単体テストで補う方法を提案する．過小近似検知の網羅度と単体テスト実行の網羅度は，産業界で高信頼ソフトウェアのテスティングに用いる MCDC 基準にしたがって測定する．MINIX を用いて提案方法の実験を行い，BMC の過小近似箇所を MCDC 基準で検知できることと，BMC と自動生成した少数のテストケースによる単体テストを合わせ

て MCDC 基準を 100% 達成できることを示した．

プログラムの事前・事後条件は入力データと期待結果と見なせるので，事前・事後条件の定義は単体テストの計画に相当する．また，BMC を用いたモジュラー検証は，単体テスト実施の自動化を実現する．さらに，単体テストと共通のカバレッジ基準を用いた BMC の過小近似検知と，BMC を事前・事後条件から生成したテストケースによる単体テストで補完する方法は，従来の単体テスト評価に相当する．本研究の提案方法は全体として，従来の単体テスト作業（計画・実施・評価）を効率化する．形式検証技術の産業界への移転を容易にする方法といえる．

Abstract

As the software technologies are extensively used these days, many advanced features are implemented with such technologies in consumer electronics or systems to constitute the social infrastructure. Since the trouble with the software-intensive systems becomes major issues of public concern, their quality is more important than ever. Program testing technologies are the primary means of software quality assurance in industry. Testing activities take more than half of the software development period. To achieve higher quality in software, new technologies are desirable.

Logic model checking methods are such that the design of a target system is automatically verified. Given a finite state transition system and a temporal logic property, the methods use the exhaustive search algorithms of the state space to determine if the property holds or not. The model checking techniques consume computing resources exponentially with the size of the target system. This scalability problem is called "state space explosion". Bounded model checking (BMC) technique alleviates the problem by limiting the scope of exhaustive search. Software model checking is an approach to apply model checking techniques including BMC to program verification. Since the approach is fully automated, it is suitable for use in industry. However, the state space explosion still occurs even for BMC when it is applied to practical programs instead of system designs. Moreover the relation between software model checking techniques and conventional program testing techniques is not clear. These issues must be solved in order to introduce software model checking approach into industry.

Modular verification is another approach of program verification. Programmers write module specifications which consist of pre-/postconditions of functions and invariants of global variables. A target program is statically checked if it conforms to module specifications. Each module but not the whole program is verified at a time,

thus programs of large size can be verified without the state space explosion.

As a baseline of our study, we proposed a new technique which integrates BMC technique with the notion of modular verification. The introduction of the modular verification technique improves the scalability in BMC. The proposed technique transforms module specifications into code fragments using verification primitives of BMC tools and embeds them into a target program. Then BMC technique searches from initial states satisfying pre-conditions and invariants to post-states of the target program, and checks that the post-states satisfy post-conditions and invariants. In the experiment using industrial programs, our technique detected errors which are revealed in unit testing conducted independently of the experiment.

Our baseline technique has weaknesses from either modular verification or BMC. Due to lack of global information along a call sequence, modular verification technique misses errors or gives false alarms. Due to under-approximation in the transformation from a target program to a finite state space, BMC technique misses errors. We propose solutions to these issues.

C programs have situations of module re-entrance or of indirect calls through pointers to functions. The situations relate to call sequences.

Module re-entrance is a situation that a module calls another module and the latter calls back the former. The situation frequently occurs in the callback-style programs. When a unit of verified module is a function, modular verification techniques miss the violation of module invariant at a re-entrant location. For this module re-entrance problem, we regard a file as a module and devised a notation to restrict the scope of invariants to a file of their definitions. Besides the file-scoped invariants, we use the in-line expansion feature of BMC tools to expand a verification unit from a function to a call sequence. We conducted an experiment and showed that our approach could accurately detect the file-scoped invariant violation at the re-entrant location.

As for indirect calls, the actually called function is determined at run-time when its address is assigned to a function pointer. It is difficult for static checkers to identify the function. Furthermore the calling and the called programs are sometimes separately developed. Codes or pre-/post-conditions of the latter cannot be used in checking of the former. We devised a notation to define pre-/post-conditions of function pointers. We also proposed two-step verification approach; modular verification using function pointers' pre-/post-conditions and substitutability checking between function pointers and actually called functions. The substitutability checking is based on substitutability of super-/sub-types in the object-oriented programming. In an experiment using

MINIX, an open-source OS, we showed that false alarms related to function pointers were reduced. We also succeeded in detection of an unknown bug related to an indirect call.

BMC techniques transform a target program into a finite state transition system without abstraction and perform accurate static analysis. However, under-approximation introduced in transformation eliminates some paths in the target program from the state space. BMC techniques miss errors on the eliminated paths. It is mandatory to detect under-approximated codes and to check them with other means. We devised a technique to automatically detect under-approximated locations and proposed test-case generation from function pre-/post-conditions. The coverage of both the under-approximation detection and program unit testing is measured with MCDC coverage criteria widely used in industry. We conducted an experiment using MINIX and showed that our technique could detect locations under-approximated with BMC technique. We also showed that the proposed technique achieved 100% coverage under MCDC.

Our approach has a clear relation with conventional program testing. Pre-/post-conditions of a function are respectively regarded as input data and expected result in testing. So the definition of pre-/post-conditions corresponds to the planning of unit testing. BMC technique adopting the notion of modular verification realizes the automated execution of unit testing. The detection of codes under-approximated with BMC technique is similar to the coverage measurement in the evaluation of unit testing. Our proposals as a whole can streamline the conventional unit-testing activities (planning, execution, and evaluation). We consider that our approach is easy to be practically introduced into industry.

目次

内容梗概 i

Abstract v

第 1 章　序論 1

1.1　研究の動機 1

1.2　既存技術と課題 1

1.3　研究の概略 3

1.3.1　BMC を用いたモジュラー検証 3

1.3.2　モジュラー検証に起因する課題とその解決策 4

1.3.3　BMC に起因する課題とその解決策 4

1.4　研究の主要な成果 5

1.5　論文の構成 6

第 2 章　背景技術 7

2.1　ロジック・モデル検査 7

2.1.1　モデル検査の発展と課題 7

2.1.2　ステートレスな探索 9

2.1.3　抽象化 9

2.1.4　有界モデル検査 11

2.2　モジュラー検証 15

2.3　有界モデル検査ツール VARVEL 18

第 3 章 VARVEL におけるモジュラー検証 25
3.1 背景 25
3.2 実現方式 26
3.3 実験 30
3.4 考察 32
3.5 関連研究 32

第 4 章 モジュール再入に関する検査 35
4.1 背景 35
4.2 問題 36
4.3 検査の方法 39
4.4 実験 44
4.5 考察 46
4.6 関連研究 48

第 5 章 関数ポインタに関する検査 51
5.1 背景 51
5.2 検査方法 54
5.2.1 仮仕様を用いたモジュラー検証 55
5.2.2 実仕様と仮仕様の置換可能性検査 58
5.3 実験 60
5.3.1 事前・事後条件のみを用いたモジュラー検証 61
5.3.2 不変条件・仮仕様を用いたモジュラー検証 61
5.3.3 仕様の置換可能性検査 66
5.4 考察 66
5.5 関連研究 67

第 6 章 ソフトウェアモデル検査とテストケース生成の統合 69
6.1 背景 69

6.1.1 テスト技術とモジュラー検証 70
6.1.2 近似の問題 75
6.2 方式 76
6.2.1 処理の概要 77
6.2.2 近似導入の検知方法 81
6.2.3 仕様からのテストケース生成 87
6.3 実験 91
6.4 考察 94
6.5 関連研究 95

第 7 章 結論 97
7.1 成果 97
7.2 今後の課題 98

謝辞 101

参考文献 103

図目次

2.1 有界な経路における 2 つの場合 11
2.2 C プログラムの有界モデル検査のためエンコーディングの例 14
2.3 F-Soft/VARVEL の処理の流れ 18
2.4 アサーション違反を起こすプログラムの例 22
2.5 VARVEL の検査結果の画面例 23
2.6 NULL ポインタ参照を起こすプログラムの例 24

3.1 DbC 仕様のプログラムコードへの変換方法 27
3.2 DbC 仕様のプログラムコードへの変換の例 28
3.3 事後条件違反を起こすプログラム例 31

4.1 オブジェクト再入の例 (C#) 36
4.2 モジュール再入の例 (C) 38
4.3 モジュラー検証による再入時の不変条件の検査 40
4.4 ファイル不変条件の例 42
4.5 モジュールを拡大した検査の例 43
4.6 モジュール再入におけるコールシーケンスを検証単位とする検証結果 . . 45
4.7 モジュール再入におけるファイル不変条件を用いたコールシーケンスを検証単位とする検証結果 45

5.1 関数ポインタを用いるプログラムの例 52
5.2 動的にロードした関数を呼び出すプログラムの例 53

5.3 仮 DbC 仕様の記述例 56
5.4 プリミティブへの変換例 56
5.5 関数ポインタを介した関数呼出しを行うプログラムを検査するアルゴリズム 57
5.6 実・仮 DbC 仕様の置換可能性を検査するアルゴリズム 58
5.7 実・仮 DbC 仕様の置換可能性検査スクリプトの例 59
5.8 不具合を含むソースコードの例 64
5.9 MINIX のデバイスドライバの要求応答メッセージ 65

6.1 malloc を 2 回呼ぶプログラムの例 71
6.2 ホワイトボックス・テストの観点のテストケース例 72
6.3 ブラックボックス・テストの観点のテストケース例 72
6.4 分岐カバレッジ基準を満たしても不具合のあるプログラムの例 74
6.5 処理の流れ 77
6.6 有界モデル検査をテストケース生成で補うアルゴリズム 79
6.7 過小近似の影響を受けるプログラム例 81
6.8 未探箇所を検知するためのトラップ入りプログラムの例 86
6.9 テストケース生成用スクリプトの例 89
6.10 テストプログラムの例 90

表目次

2.1 LTL 式の命題論理式への変換 12
2.2 F-Soft/VARVEL が検出できる典型的なプログラミングエラー 18

3.1 VARVEL の DbC 記法 26
3.2 DbC 基本機能の実験結果 30

4.1 モジュール再入時の不変条件違反の検出 44

5.1 仮 DbC 仕様の記法 55
5.2 仮 DbC 仕様のソースコードへの変換ルール 57
5.3 実験対象としたソースコード 60
5.4 事前・事後条件のみを用いたモジュラー検証の結果 61
5.5 不変条件と仮 DbC 仕様を追加したモジュラー検証の結果 62
5.6 実 DbC 仕様と仮 DbC 仕様の置換可能性検査の結果 66

6.1 トラップ挿入のための整形ルール 83
6.2 モデル検査によるカバレッジの例 87
6.3 モデル検査とテストによるカバレッジの例 91
6.4 MINIX への提案方法の適用結果 92
6.5 MINIX を対象としたモデル検査とテスト実行によるカバレッジ 93

第 1 章

序論

1.1 研究の動機

ソフトウェアは一般消費者が用いる様々な製品や社会基盤システムに浸透している．製品やシステムの多機能化・高機能化が進む中で，多くの機能がソフトウェアにより実現されている．社会問題となるシステム障害がソフトウェアに起因することも多く，プログラムの更なる品質向上が求められている．産業界では，プログラムの品質をテスティングによって確保してきた．テスティングでは，入力データを与えてプログラムを実行し，その出力データを期待結果と比べて，プログラムの機能・振る舞いが正しいことを確認する．1 組の入力データに対して，プログラムの 1 つの経路 (path) を実行する．より多くの経路を調べるには，より多くの入力データを与えて，プログラム実行を繰り返す必要がある．近年は，ソフトウェア開発期間の半分以上をテスティングに費やしている [14] ．プログラムの品質を確保する新しい技術が必要である．

1.2 既存技術と課題

プログラムの品質向上の手段として，形式手法 (formal methods) に期待が集まっている．形式手法は，数理論理や集合といった数学的な概念を基盤として，システムやソフトウェアの構造や機能・振る舞いを正確に記述し，厳密な検証を行う手法の総称である [69] ．産業界での形式手法の利用には，大きく 2 つの使い方がある．一方は開発上流における

利用であり，他方はプログラムの検査への応用である [71] .

開発上流における利用では，要求仕様や設計情報の表現と解析に形式手法を適用する．「構築からの正しさ（correct by construction)」という理想的な姿を目指す使い方である．仕様・設計情報を検証しながらシステムやソフトウェアの構築 (construction) を進め，完成した仕様や設計情報に誤りがない (correct) ことを保証する．形式手法の理想的な使い方ではあるが，プログラムと同程度の記述量を必要とする [1] ．産業界への導入には敷居が高い．

プログラム検査への形式手法の応用は，プログラムという最終成果物を作った後に行うことから「事後検証（a posteriori verification)」と呼ばれる．プログラムコードの数理論理表現への変換からプログラムの正しさの検査までを自動で行うプログラム自動検証が理想である．プログラム自動検証は，プログラムを検査の対象とすることから，従来の品質向上手段であるプログラム・テスト技術を直接的に補う使い方といえる．実際に，形式手法を用いたプログラム検査ツールの研究開発が進んでいる [9] [12] [16] [25] [27] [36] [38] [40] [49] .

ソフトウェアモデル検査は，形式検証の 1 手法であるロジック・モデル検査を用いたプログラム自動検証である [55] ．プログラムを有限状態遷移系に変換し，1 回の検査で，有限状態空間上の実行しうる全経路を探索する，網羅探索によって，不具合の見逃しがないことが特長である．しかし，モデル検査には，「状態爆発」と呼ばれるスケーラビリティの問題があり， 状態数や遷移の組合せといった検査対象の規模が大きくなると，メモリ消費量や計算時間が指数的に増える．状態爆発が起こると，不具合の有無を判定できない．ソフトウェアモデル検査では，検査対象の大規模化に対応する技術の確立が実用化のポイントである．状態爆発の問題を防ぐために，検査対象システムや検査条件の特徴を考慮した様々な手法が提案されてきた．有界モデル検査（bounded model checking. BMC）は従来提案手法の一つであり，有限状態空間の探索範囲を限定することにより，不具合を効率よく発見する [17] ．しかし，実用的な C プログラムの場合，BMC でも状態爆発の問題が発生することが知られている．また，ソフトウェアモデル検査は，入力データに対して出力データが期待どおりであることを調べるプログラム単体テストとの関係も明らかではない．

モジュラー検証は，プログラム全体の検査をモジュールごとの検査に分けて行う考え方である．堅牢なソフトウェアを構築するための Design by Contract (DbC) [64] の考え方を取り入れて，プログラム中に関数・手続きの事前・事後条件や大域変数の不変条件といった DbC 仕様を定義する．検証ツールは，関数呼び出しがあると，呼ばれる関数のプログラムコードの代わりに利用者が定義した DbC 仕様を用いる．DbC 仕様を境としてプログラムをモジュールに分け，モジュールごとに検査を行うので，プログラム規模の影響を受けにくい．また，事前・事後条件がそれぞれプログラム単体テストの入力データと期待結果に相当することから，テスティングとの親和性がある．モジュラー検証を行う拡張静的チェッカが開発されている [36] [38] ．しかし，拡張静的チェッカは網羅的な検査を目的とせず，不具合を見逃すことも多い．

1.3　研究の概略

本研究では，逐次 C プログラムを対象として，安全性に関する検査条件を調べる BMC を採用する．まず，状態爆発の問題を避けるために，モジュラー検証の考え方を BMC に導入し，BMC を用いたモジュラー検証 (DbC に基づく BMC．DbC ベース BMC) を実現する．次に，モジュラー検証に起因する課題を明らかにし，解決方法を考案し，その効果を実験により実証する．さらに，BMC に起因する課題を明らかにし，解決方法の考案と実験による効果の実証を行う．

1.3.1　BMC を用いたモジュラー検証

本研究では，状態爆発の問題を避けるために，モジュラー検証の考え方を BMC に導入する．人手で C プログラムの DbC 仕様を記述するにあたり，産業界の開発者にとって理解しやすい記法を提供した．そして，DbC 仕様を BMC で探索できる検査条件に変換し，DbC 仕様を境に C プログラムをモジュールに分割する検査方法を考案した．これらの記法と検査方法により，BMC を用いたモジュラー検証である，DbC に基づく BMC（DbC ベース BMC）を実現した．DbC ベース BMC は，関数ごとの検査を実現するので，C プログラム全体を探索する従来の BMC における状態爆発の問題を軽減する．

1.3.2　モジュラー検証に起因する課題とその解決策

DbC ベース BMC には，モジュラー検証の考え方に起因する課題がある．産業界の C プログラムでは，大域変数を介した関数間でのデータのやり取りや，関数ポインタによる間接呼出しが行われる．C プログラムでは，ある機能を実現する複数の関数をまとめてモジュールと呼ぶ．モジュールの最小単位は関数である．関数ごとのモジュラー検証では大域変数がコールシーケンスに跨って影響するような検査条件を調べにくい．この問題は大域関数の値を不変条件を用いて限定することにより緩和できる．しかし，大域変数のスコープに合わせて不変条件のスコープもプログラム全体であることから，あるモジュールが呼び出した別モジュールの関数が再び元のモジュールの関数を呼び返すモジュール再入の状況では，再入箇所での不変条件の検査を適切に行うことができない．関数ポインタについては，間接的に呼ばれる実際の関数は実行時に決まることから，静的なプログラム自動検証が難しい．GUI フレームワークやデバイスドライバなど関数ポインタを用いたコールバック型プログラムでは，呼ばれる側と呼び出し側が別々に開発されるため，コールシーケンス全体をまとめて検査することも困難である．本研究では，不変条件のスコープを限定する DbC の記法と検査対象モジュールをコールシーケンスに拡張した検査方法を提案し，再入状況における不変条件の検査を可能とした．また，関数ポインタに事前・事後条件（仮 DbC 仕様）を付与する DbC の記法と，仮 DbC 仕様を用いたモジュラー検証の方法と，関数ポインタと実際に呼ばれる関数の置換可能性を検査する方法を提案し，関数ポインタによる間接呼出しが関係するプログラムの自動検証を可能とした．

1.3.3　BMC に起因する課題とその解決策

DbC ベース BMC では，BMC に起因する課題もある．本研究では，BMC の一般的な限界についての問題分析を行い，DbC ベース BMC をテストケースの自動生成によって補う新しい方法を考案した．BMC では検査対象 C プログラムに近似変換を施すことで有限状態遷移系を得る．元のプログラムよりも変換後のプログラムの経路が多くなる過大近似では，元の C プログラムには無い不具合を有限状態空間上で見つける「誤警告」「見か

けの反例」という問題が起こる．元のプログラムよりも変換後のプログラムの経路が少なくなる過小近似では，元の C プログラムにあったバグが有限状態空間では消える「不具合の見逃ごし」という問題が起こる．

過大近似に起因する問題では，反例から得た入力データを与えてプログラムを実行し，不具合が再現しなければ，見かけの反例と判断できることが知られている．一方，過小近似に起因する問題については，その存在自体が陽に言及されることが少ない．

そこで，本研究では，DbC ベース BMC を応用して，過小近似が行われたことを自動的に検出する方法を考案した．さらに，過小近似によって検査されないプログラム箇所をピンポイント的に検査するテストケースを自動生成し，これを用いたテスト実行によって BMC の問題点を補う方法を着想した．上記の自動検出ならびにテスト実行について，産業界で使われているホワイトボックス・テストの網羅性メトリクス MCDC (modified condition decision coverage) を共通に用いることで，自動化可能な方式を考案した．

1.4　研究の主要な成果

本研究の第 1 の成果は，DbC ベース BMC を明確に導入したことである．産業界のプログラムに適用し，開発者が DbC 仕様を記述できること，実験とは独立して行われたプログラム単体テストで発見された不具合を，DbC ベース BMC でも発見できることを示した [44] .

本研究の第 2 の成果は，スコープを限定した不変条件の記法とモジュール再入時の検査の方法を確立したこと [44] ，C プログラムで頻繁に使われる関数ポインタを考慮して DbC 仕様の表現と検証の方法を確立したことである [46] ．関数ポインタに関する提案方法の有効性をオープンソース OS である MINIX に適用し，未発見のバグを見つけることに成功した．これは関数ポインタの使用に起因するものであって，他の既存 BMC ツールでは見逃ごす誤りである．

本研究の第 3 の成果は，DbC ベース BMC とテストケース自動生成を組み合わせて，プログラムの検査網羅度を達成する方法を確立したことである [48] ．MINIX を題材とした適用実験を行い，DbC ベース BMC によるモデル検査と自動生成した少数のテスト

ケースを用いたテスト実行によって，MCDC 基準が満たされることを確認した．過小近似箇所の検知とテスト実行に共通の検査網羅性基準を用いることにより，自動検証の技術とプログラム・テスティングの技術との関係を明らかにすることに成功した．

1.5　論文の構成

本論文は次の構成をとる．第 2 章では，プログラムの静的検査における基本的な技術であるソフトウェアモデル検査とモジュラー検証について述べる．第 3 章では，本稿で用いるツールとして，有界モデル検査を用いたモジュラー検証 (DbC ベース BMC) を行う VARVEL を取り上げ，VARVEL における DbC 仕様の記法と検査の実現方法を説明する．モジュラー検証では，コールシーケンスに跨った情報の不足や，呼ばれる関数が検査時に特定できない間接呼出しによって，誤った検査結果を得るという問題がある．これらの問題について，それぞれ第 4 章と第 5 章で，DbC 仕様の記法と検査方法を提案し，実験によって検査が実施できることを示す．ソフトウェアモデル検査はテスティングに代わる有用な技法であるが，プログラムを有限状態遷移系に変換する際に近似を導入する．近似の導入によって，誤警告や不具合の見逃ごしといった問題が生じる．第 6 章では，網羅性メトリクス MCDC を考慮して近似の導入を自動検知し，DbC 仕様からのテストケース自動生成によって，ソフトウェアモデル検査を補完する方法を提案する．提案方法により MCDC を満たす検査が実現できる．最後に，第 7 章で，一連の成果と今後の課題について述べる．

第 2 章

背景技術

ソフトウェアの信頼性向上に関心が寄せられている．従来からの信頼性向上手段であるプログラム・テスト技術は，プログラム実行に依るので，１つのテストケース毎に１つの経路しか検査せず，網羅度が低い．信頼性向上の新たな手段として，ソフトウェアモデル検査といった，プログラムの静的解析手法が望まれている．しかし，産業界では，信頼性の基準はプログラム・テスト技術で与えられており，静的解析手法による検査とテストの関係を論じる必要がある．

2.1 ロジック・モデル検査

2.1.1 モデル検査の発展と課題

ソフトウェアあるいはプログラムの信頼性や安全性，すなわち，ディペンダビリティの問題に技術的な解決策を与える形式検証の技術が注目を集めている．なかでも，ロジック・モデル検査は自動検証法として最も重要な技術のひとつに数えられている [70] .

ロジック・モデル検査では，対象を有限状態遷移系として表し，検査したい性質 (プロパティ) を時相論理で表現する．有限状態空間を網羅的に探索して，プロパティの成立を判定する．状態系列（経路，path）を考慮して，初期状態から到達可能な全ての状態を探索するので，探索が完了した場合には，パス網羅基準を満たす網羅度がある．プロパティに反する状態あるいは経路を見つけると，反例を出力する．モデル検査の反例は，初期状

態からプロパティ違反となった状態に至る経路と経路上の各状態の情報 (変数の値) を含む．従来のテスト技術に比べて極めて高い網羅性があることに加えて，具体的な変数値を含む反例がデバッグ作業に役立つことから，モデル検査はテスト技術を補う手段として有望である．

しかし，モデル検査には，状態爆発と呼ばれる本質的なスケーラビリティの問題がある．状態や状態遷移が増えると，計算量が指数的に増える．状態や状態遷移が大規模な対象では，CPU 時間やメモリといったリソースが不足し，不具合の有無を判定できないことがある．

ロジック・モデル検査は，当初はハードウェアや通信プロトコルの検証に適用されてきた．近年，計算機ではハードディスクの代わりにメモリディスクが，自動車ではエンジンの代わりに電気モーターが使われることが増えており，その制御をソフトウェアが担っている．このようにソフトウェアの重要性が増す中で，その品質の保証手段として，ロジック・モデル検査によるソフトウェア検証への期待はますます大きくなるであろう．すでに，自動車関連ソフトウェア，マイクロコントローラ，Linux デバイスドライバ，ファイルシステム，ネットワークフィルタ，プロトコルスタック，サーバアプリケーション，フラッシュストレージ基盤など様々なターゲットシステムへの適用事例が報告されている [57]．

ソフトウェアモデル検査は，ロジックモデル検査 [23] のプログラムの検査への適用方法である．2 進数で状態を考えるハードウェアに比べて，プログラムでは，整数や構造をもったデータを扱うため，状態の数が格段に多い．プログラムでは，出現する各変数の値の組合せの１つ１つが状態である．N 個の 32 ビット整数型変数が出現するプログラムでは，プログラム上のある箇所における状態の数は最大で 2 の 32 乗 (約 42 億) の N 乗となる．また，設計の検証では，状態爆発を軽減するために，人手で，対象システムの設計記述から関心のある情報だけを抽出して，モデル検査用の形式記述を書くことが行われてきた [52]．一般に，プログラムの記述は設計よりも量が多く，複雑である．プログラムから検証用の形式記述を人手で作ることは現実的ではないので，ソフトウェアモデル検査ではプログラムから有限状態遷移系への自動変換を行う．その際，スケーラビリティの問題をいかに軽減するかは重要な工夫である．

2.1.2　ステートレスな探索

プログラムの検査において，スケーラビリティの問題を軽減する工夫の一つとして，ステートレスな探索 (state-less search) アルゴリズムがある．ステートレス探索では，探索済みの状態を記憶しないことによって，状態爆発を軽減する [40]．冗長な探索を行わないので，効率的な検査を行える．VeriSoft [41] は，状態を記憶しない探索アルゴリズム (state-less search) を用いたモデル検査ツールである．並行処理の C プログラムを探索し，デッドロックやデッドコードやユーザ指定表明に対する違反といった安全性に関する不具合を検出する．しかし，ステートレス探索は，現在探索中の経路の情報しかもたないため，安全性 (safety) のプロパティは検査できるが，活性 (liveness) については検査できない．

2.1.3　抽象化

述語抽象化 (predicate abstraction) は，変数値の制約を表す述語を考え，述語の値を保持するブール型変数で元の変数を置き換えることにより，状態数を削減し，状態爆発を軽減する [42]．抽象化において，どの変数のどのような制約を述語とすべきかは自明ではない．

初期の Java PathFinder [49] (JPF) は述語抽象化を用いたモデル検査ツールである．並行あるいは逐次 Java プログラムを入力とし，デッドロックやユーザ指定表明に対する違反といった安全性に関する不具合を検出する．プログラマは，JPF が提供する抽象化用のクラスを用いて，抽象化に用いる述語やどのように抽象化を行うかをソースコードに明記する．述語の抽象化と洗練をプログラマが行うのは負担が重い．なお，JPF は自動検査からテストケース自動生成に方向を変えている [82]．

Ball らは，プログラムの検査において，ループや条文分岐の制御式を述語として抽象化を自動で行うことを提案した [7]．述語抽象化では，見かけの経路 (infeasible path）が増える．述語抽象化を用いたモデル検査では安全性 (safety) を検査する．安全性のプロパティについて，見かけの経路上では起こるが，実際の経路では起こり得ない不具合を検出

することがある．誤警告 (false alarm, spurious counter-example) と呼ばれる問題である．反例を利用した抽象化の洗練 (CEGAR) [24] は，見かけの経路を自動的に排除する手法である．モデル検査によって不具合を検出した場合に，その反例に現れる経路を記号実行によって走査する．不具合を再現できない場合は，見かけの経路上の不具合であると判断する．見かけの経路を探索しないようにするための新たな述語を導入し，モデル検査を再実行する．CEGAR は有用な考え方であるが，抽象の洗練が収束しないことがある．

SLAM [9] と BLAST [50] は述語抽象化と CEGAR に基づく述語の洗練を自動で行うモデル検査ツールである．

SLAM は，Windows OS のデバイスドライバ専用の検査ツールであり，逐次 C プログラムが OS のデバイスドライバ用 API を正しく使っていることを検査する．デバイスドライバが API を誤って使うと，OS がクラッシュすることがあるので，この検査は OS の信頼性向上のために重要である．API の仕様 (プロパティ) は，ツール提供者によって SLIC 言語を用いて予め記述されて，ツールに組み込まれている．SLAM は，プロパティに基づいて述語抽象化を自動的に行い，元のプログラムをブーリアンプログラムに変換し，モデル検査を行う．さらに，反例が見かけかどうかを判定し，見かけの場合には反例が起きる経路を除外するような述語を追加する [8]．SLAM のブーリアンプログラムは，プッシュダウンオートマトンと同じ表現力を持つことから，SLAM は再帰関数も扱える [6]．

BLAST は，逐次 C プログラムが API を正しく使っていることを検査する．検査者は，API が正しく使われていることを示すプロパティモニターコードを C プログラムの関数として記述する．ロックの獲得や解放といった API を使う順番をプロパティモニターとして記述する．BLAST は，対象プログラムにおける API の呼び出しに合わせて，プロパティモニター上で想定どおりの状態遷移が起こることを，モデル検査により検査する．CEGAR により反例が見かけの経路上で起こる不具合を示すと判った場合にのみ，述語抽象化を行う点が特徴である [51]．

述語抽象化では，元のプログラムには存在しない見かけの無限ループが導入された場合に，活性 (liveness) のプロパティ違反を見逃ごすことがある．Pnueli らは，活性のプロパティを検査できるランキング抽象 (ranking abstraction) を提案した [56]．ランキング抽

象では，プログラムの特定の箇所において値が減少し，他の箇所では値が増えず，かつ，値域が有限である関数 (ランキング関数) を用いた進行性モニタ (progress monitor) を導入し，進行性モニタと検査対象とを合わせて，検査したい性質が成り立つかどうかを調べる．ある性質について，ランキング関数の値が，値域が有限であるのに，減少し続けるようであれば，その性質が成り立たないことが判る．Terminator はランキング抽象を用いたモデル検査ツールであり，プログラムのループを調べてランキング関数を自動的に決める．Terminator は逐次 C プログラムが停止することを調べる [28] ．

2.1.4　有界モデル検査

有界モデル検査 (bounded model checking. BMC) は，モデル検査の探索範囲を一定の深さまでに限定し，不具合を効率よく検出する手法である [18] [74] ．近年，性能向上の著しい充足可能性判定アルゴリズム (SAT) を用いることにより，不具合を見つける [80] [74] ．BMC では，探索範囲外に不具合が存在するかどうかは判らない．しかし，状態遷移グラフや二分決定グラフ (BDD) により全ての状態を記憶することはせず，探索中の経路の情報しか保持しないので，効率的に不具合を検出できる [17] ．

図 2.1　有界な経路における 2 つの場合
文献 [18] から引用.

BMC において，探索範囲を k ステップまでに限定した場合の経路には，図 2.1 に示す 2 つの場合がある．(b) は，状態 S_k から $S_l(l = 0..k)$ へのループがある場合であり，対象が無限長の経路を含む場合にも，その振舞いを長さ (経路上の遷移数)k の有限状態遷移系で表現できる．(a) は，ループが無い場合であり，元の対象における状態 S_{k+1} 以降の振舞いを長さ k の有限状態遷移系では表現できない．

BMC では，長さ k の有限状態遷移系を表す命題論理式 $[\![M]\!]_k$ と長さ k の有限状態遷

移系において状態 s_i からの経路に関する時相論理式を表す命題論理式 $[\![f]\!]_k^i$ との積である $[\![M,f]\!]_k$ を満たす変数値の割り当てを探す．$s_i(i=0..k)$ を経路の i 番目の状態，状態 s が初期状態にあることを $I(s)$，s_i から s_j への状態遷移を $T(s_i,s_j)$ とすると，経路の長さ k の有限状態遷移系の構造は次の命題論理式となる．

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \qquad \text{(出典：文献 [18] Definition 10)}$$

表 2.1　LTL 式の命題論理式への変換

ループなしの場合 (図 2.1(a))	ループありの場合 (図 2.1(b))
$[\![p]\!]_k^i := p(s_i)$	${}_l[\![p]\!]_k^i := p(s_i)$
$[\![\neg p]\!]_k^i := \neg p(s_i)$	${}_l[\![\neg p]\!]_k^i := \neg p(s_i)$
$[\![f \wedge g]\!]_k^i := [\![f]\!]_k^i \wedge [\![g]\!]_k^i$	${}_l[\![f \wedge g]\!]_k^i := {}_l[\![f]\!]_k^i \wedge {}_l[\![g]\!]_k^i$
$[\![f \vee g]\!]_k^i := [\![f]\!]_k^i \vee [\![g]\!]_k^i$	${}_l[\![f \vee g]\!]_k^i := {}_l[\![f]\!]_k^i \vee {}_l[\![g]\!]_k^i$
$[\![\mathbf{G}f]\!]_k^i := \mathbf{false}$	${}_l[\![\mathbf{G}f]\!]_k^i := \bigwedge_{j=min(i,l)}^{k} {}_l[\![f]\!]_k^j$
$[\![\mathbf{F}f]\!]_k^i := \bigvee_{j=i}^{k} [\![f]\!]_k^j$	${}_l[\![\mathbf{F}f]\!]_k^i := \bigvee_{j=min(i,l)}^{k} {}_l[\![f]\!]_k^j$
$[\![\mathbf{X}f]\!]_k^i := if\ i<k\ then\ [\![f]\!]_k^{i+1}\ else\ \mathbf{false}$	${}_l[\![\mathbf{X}f]\!]_k^i := {}_l[\![f]\!]_k^{succ(i)}$
$[\![f\mathbf{U}g]\!]_k^i := \bigvee_{j=i}^{k}([\![g]\!]_k^j \wedge \bigwedge_{n=i}^{j-1}[\![f]\!]_k^n)$	${}_l[\![f\mathbf{U}g]\!]_k^i := \bigvee_{j=i}^{k}({}_l[\![g]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[\![f]\!]_k^n) \vee$
	$\bigvee_{j=l}^{i-1}({}_l[\![g]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![f]\!]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[\![f]\!]_k^n)$
$[\![f\mathbf{R}g]\!]_k^i := \bigvee_{j=i}^{k}([\![f]\!]_k^j \wedge \bigwedge_{n=i}^{j}[\![g]\!]_k^n)$	${}_l[\![f\mathbf{R}g]\!]_k^i := \bigwedge_{j=min(i,l)}^{k} {}_l[\![g]\!]_k^j \vee$
	$\bigvee_{j=i}^{k}({}_l[\![f]\!]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[\![g]\!]_k^n) \vee$
	$\bigvee_{j=l}^{i-1}({}_l[\![f]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![g]\!]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[\![g]\!]_k^n)$

$[\![\cdot]\!]_k^i$ 長さ k の経路における i 番目の状態についての変換.
${}_l[\![\cdot]\!]_k^i$ k 番目から l 番目への状態遷移を持つ長さ k の経路における i 番目の状態についての変換.
p 命題論理式.　f,g 線形時相論理 (LTL) 式.　$\mathbf{G},\mathbf{F},\mathbf{X},\mathbf{U},\mathbf{R}$ 時相演算子.
$k,l,i,j,n \in \mathbb{N}$，$l,i \leq k$.　$succ(i) := i+1 (i<k$ の場合$)$，$succ(i) := l (i=k$ の場合$)$
(出典：文献 [18] Definition 11-13)

線形時相論理 (LTL) 式から命題論理式への変換 $[\![\cdot]\!]$ は，表 2.1 のルールに従う．とくにループがある場合に，LTL 式から導出した命題論理式が多数の項を持つ複雑な式になることがわかる．たとえば，いつか f が成り立つことを表す $[\![\mathbf{F}f]\!]$ は各状態 s_i における f の論理和であり，探索の深さ k に応じて節が増える．

有限状態遷移系の構造と LTL 式とを合わせて，充足可能性判定 (SAT) 問題として解く

べき式 $[\![M, f]\!]_k$ は，ループがある場合，次のようになる．

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge \left(\left(\neg(\bigvee_{l=0}^{k} T(s_k, s_l)) \wedge [\![f]\!]_k^0\right) \vee \bigvee_{l=0}^{k}\left(T(s_k, s_l) \wedge {}_l[\![f]\!]_k^0\right)\right)$$

ループが無い場合は，より簡潔な次の式で済む．

$$[\![M, f]\!]_k := [\![M]\!]_k \wedge [\![f]\!]_k^0$$

2 進数で表現されるハードウェアに比べて，整数や構造を持つデータを扱うプログラムは状態や遷移の数が多い．そこで，BMC をプログラムの検査に適用する場合，まず，対象プログラムをループの無い有限状態遷移系 $C = I(s_0) \wedge (\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}))$ に変換する．プログラム中のループはボディを固定回数だけ繰り返す逐次処理に変換する．たとえば，固定回数を 2 回とすると，制御式 E とループボディ S をもつループ文 $while\ (E)\ S$ を逐次処理 $if\ (E)\ \{\ S;\ if\ (E)\ \{\ S;\ assert(\neg E);\ \}\ \}$ に変換する．検査したい性質 (プロパティ) を P とすると，プログラムが実行された際にプロパティを満たすことは $C \Rightarrow P$ となる（$\Rightarrow$ は論理の含意）．BMC では，その否定の式 $C \wedge \neg P$ を満たす変数値の割当を SAT により探す．見つかった変数値の割当は，対象プログラムがプロパティを満たさないことを示す反例，即ち，ある初期状態からプロパティを満たさない状態に至る経路の一例である．見つからなければ，対象プログラムはプロパティを満足する．例えば，プロパティとして安全性 $\mathbf{G}P$ を調べる場合，BMC には LTL 式として検査したい性質の否定 $\mathbf{F}\neg P$ を与えることになる．SAT 問題として解くべき式 $\phi^{(k)}$ は次のようになる．

$$\phi^{(k)} = I(s_0) \wedge (\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})) \wedge (\bigvee_{j=0}^{k} \neg P(s_j))$$

ある状態 s_j でのみ判断すればよい性質を調べる場合には，SAT 問題 $\phi^{(k)}$ はさらに簡潔な式で済む．たとえば，プログラム中に記述されたアサーション P_a は，記述箇所に相当する状態を s_a とすると，次のように書ける．

$$P(s_j) := (s_j = s_a) \rightarrow P_a$$

アサーションを検査するための SAT 問題は，次の式となる．

$$\phi^{(k)} = I(s_0) \wedge (\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})) \wedge (s_a \wedge \neg P_a)$$

プログラム検査への BMC の適用では，上述のように，プログラムをループのない有限状態遷移系に変換する，プロパティをある状態への到達時にのみ調べればよい性質に限定する，といった工夫を行い，SAT 問題を簡潔な式にして，状態爆発を軽減する．なお，活性 (liveness．$\mathbf{F}f$) を調べる場合，BMC で解くべき SAT 問題にはその否定 ($\mathbf{G}\neg f$) を与

える．ループがない場合，この否定の式は恒偽 ($false$) であり，検査は無意味となることに注意する必要がある．

命題論理式の充足可能性を判定する SAT に対して，近年，整数や配列などを含む論理式を扱える SMT(Satisfiability Modulo Theories) [39] の研究が進み，SAT に代わって有界モデル検査に利用され始めている．SMT は一階述語論理の決定可能サブクラスを扱い [19] [59]，SAT 問題にエンコードする方法に比べて計算が高速であることが多い．なお，BMC にはランキング関数を用いて活性 (liveness) のプロパティを扱う研究がある [58]．

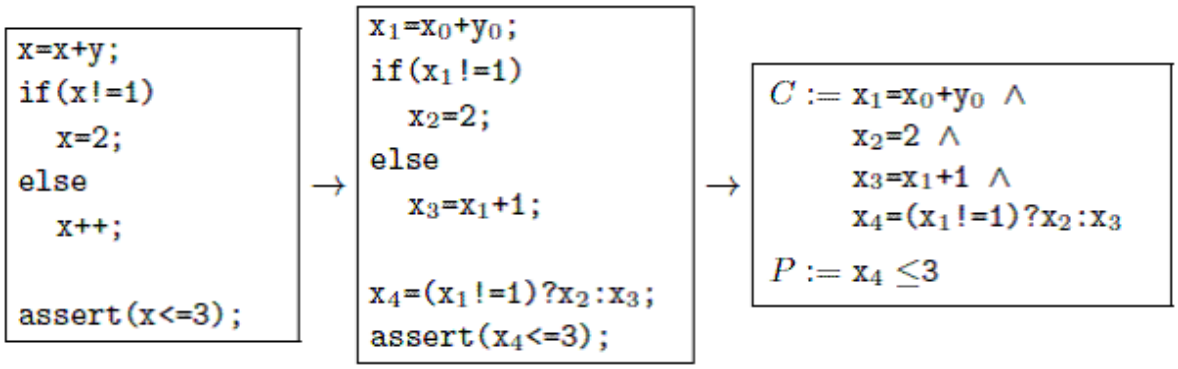


図 2.2　C プログラムの有界モデル検査のためエンコーディングの例
文献 [25] から引用.

CBMC [25] は，SAT に基づく有界モデル検査を用いたツールであり，標準的な方法と考えられている．C プログラムを入力とし，NULL ポインタ参照や配列境界違反やユーザ指定表明に対する違反といった安全性に関する不具合を検出する．入力されたプログラムを，各変数が一度のみ代入されるような静的単一代入 (SSA: Static Single Assignment) 形式に変換し，プログラムの実行を示す式 C と検査したい性質を示す式 P を生成する (図 2.2)．プログラムを有界な有限状態遷移系に変換するので，範囲外の不具合は判らない．しかし，範囲内の検査は完全に自動で行うことができる．[26]．

同様のツールに F-Soft [54] がある．F-Soft は安全性のプロパティを検査する．エラーを起こす状態への到達性 (reachability) を調べ，到達する場合にはプロパティ違反を警告する．

また，SMT に基づく有界モデル検査ツールとして SMT-CBMC [4] や ESBMC [29] がある．検査できる性質は CBMC と同様である．

本論文で用いる VARVEL [67] は F-Soft を機能強化したツールである．SAT に基づく有界モデル検査を行う．

2.2　モジュラー検証

一般に，CBMC [25] や F-Soft [54] といった BMC を用いたプログラム検査ツールは呼ばれる側の手続きをインライン展開する．インライン展開は状態爆発の問題を悪化させ，スケーラビリティを損ねる結果となる．さらにデータ・制御フローに依存した解析を行う場合，呼出し元の手続きも全て考慮する必要があり，一度の解析に必要なプログラムの規模は巨大になる．プログラムが巨大になり BMC の探索の深さを超えると，不具合の見逃ごしが発生し得る．

モジュラー検証は，プログラム全体の検査をモジュールごとの検査に分けて行うことにより，スケーラビリティの問題を解決する考え方である．各モジュールの検査では，そのモジュール自体やそこから呼ばれるモジュールのインタフェース仕様を与える必要がある．Design by Contract (DbC) の考え方を採用することによりモジュールのインタフェース機能仕様を与えることができる．DbC は元々は堅牢なソフトウェアを構築するアプローチとして提唱された [64] ．DbC では手続きの機能仕様を事前および事後条件からなる DbC 仕様として明示する．事前・事後条件はそれぞれ手続きの開始・終了時点で成立しなければならない．[66] ．

モジュラー検証では，事前条件を前提として，モジュールを実行した後に事後条件が成立することを検査する．検査対象のモジュール M が事前条件 Pre_M と事後条件 $Post_M$ を持つとき，Pre_M が成立すればモジュールの本体コード $Body_M$ を実行し，実行後の状態が $Post_M$ を満たすことを保証せねばならない．この保証すべき条件は，たとえば Dijkstra の最弱事前条件 [33] の考え方を用いると，次のように表せる [38] ($\Rightarrow$ は論理の含意．$wp(S, Q)$ は，その状態からプログラム S を実行したときに条件 Q を満たす事後状態に到達し得るような事前状態について，事前状態が満たすべき最も弱い条件を表す)．

$$Pre_M \Rightarrow wp(Body_M, Post_M) \qquad (1)$$

モジュール M から呼ばれるモジュール P が事前条件 Pre_P と事後条件 $Post_P$ を持つ

とき，呼出し元の M においては，P を呼び出す直前の状態 S で Pre_P が成り立ち，呼出し復帰後の状態 R では P 本体がどのように実行されるかに関知せずに $Post_P$ が成り立つことを保証すればよい．この保証すべき条件は次のように表せる．

$$S \Rightarrow Pre_P, \quad Pre_P \wedge Post_P \Rightarrow R \qquad (2)$$

呼出し元と呼ばれる側のモジュールを，それぞれ (1) と (2) に従って独立に検査するので，モジュラー検証が実現できる．モジュールを実行する前後の不変条件をそれぞれ Inv，Inv' として，不変条件を考慮した場合，(1)(2) はそれぞれ下記のようになる．

$$Pre_M \wedge Inv \Rightarrow wp(Body_M, Post_M \wedge Inv') \qquad (1')$$

$$S \Rightarrow Pre_P \wedge Inv, \quad Pre_P \wedge Inv \wedge Post_P \wedge Inv' \Rightarrow R \qquad (2')$$

産業界で日々行われるブラックボックスの単体テストでは，1 組の入力データと期待結果を用意する．入力データを与えてプログラムを実行し，その実行結果が期待結果に合っていることをもって，プログラムの正しさを確認する．DbC の事前条件と事後条件は，それぞれテストにおける入力データと期待結果に相当する．DbC の仕様に基づくモジュラー検証は単体テストに類似しており，プログラマにとって理解しやすい．

もともと，DbC の検査は，プログラミング言語 Eiffel で実現された．Eiffel では，プログラムとして記述された事前・事後条件を，コンパイラによって検査用のコードに変換し，実行時に検査する [66]．Eiffel のように，DbC の記法をプログラミング言語仕様の一部とする研究がある．SAL(Standard Annotation Language) は，C プログラム用の仕様の記法である．ポインタが NULL か否か，配列の要素数やバイト数といった情報を，関数のシグネチャの一部として記述する [43]．Spec#は，C#の拡張であり，C#プログラムにおいてメソッドシグネチャとして事前・事後条件を記し，クラスのメンバ変数と同様にオブジェクト不変条件を記す．これらの仕様を OS などにおける逐次プログラムが満足することを定理証明器によって検査する [12]．VCC も，Spec#同様の記法と定理証明器を用いており，並列処理プログラムである OS のハイパーバイザの検査に利用されている [27]．DbC の記法をサポートするライブラリを提供する研究もある．Code Contracts は，C#用の仕様の記法である．仕様を記述するためのクラスライブラリが提供されてお

り，プログラマはこのライブラリを用いて，C#コードの一部として仕様を記述する [11]．このようなプログラムコードの一部として仕様を記述する方法では，専用のコンパイラや実行環境が必要となる．また，本来のプログラム部分の可読性が低下するという問題がある．

単体テストに相当するモジュラー検証は拡張静的チェッカとして実現されている．ESC/Java は逐次 Java プログラムの検査ツールである [38]．プログラマは JML (Java Modeling Language) [60] に類似した記法で，Java プログラムのコメント中に DbC 仕様を記述する．ESC/Java は，まず，最弱事前条件の考え方に則って，メソッドの事後条件からメソッド開始時の最弱事前条件を算出し，最弱事前条件が成り立てばメソッドの事前条件が成り立つことを示す検証条件 (verification condition) を生成する．次に，定理証明器 Simplify [31] を用いて，検証条件が成立するか否かを調べる．Caduceus は逐次 C プログラムの検査ツールである [36]．プログラマは ACSL (ANSI C Specification Language)[13] 記法で，C プログラムのコメント中に DbC 仕様を記述する．ACSL は JML に触発されて開発された記法である．Caduceus は，プログラムを定理証明用の中間表現形式 Why に変換する．そして，Why プログラムをもとに定理証明器ごとの検証条件を生成し，定理証明器を実行する．Caduceus は中間表現を介することにより，さまざまな定理証明器に対応している．

本論文で用いる VARVEL ツールは，DbC に基づくモジュラー検証を実現しており，有界モデル検査におけるスケーラビリティの問題を軽減する．VARVEL の記法では，JML や ACSL と同様にプログラムのコメントに DbC 仕様を記述する．コンパイル時には従来のコンパイラをそのまま使うことができる．プログラムと DbC 仕様は，それぞれソースコードとコメントに分離されているので，プログラムの可読性は低下しない．事前条件などの記述内容は C 言語の式で表現できる範囲にとどめている．これにより，たとえばループといった制御に関する表現力は低くなるが，開発者が理解しやすい記述が可能となり，仕様記述の習得や仕様自体のデバッグのコストを低減できる．仕様をどこまで詳しく記述するのかは，開発者が決める．DbC 仕様の記述は労力ではあるが，実際に起こりえる不具合をツールが発見し易くなる [73]．

2.3 有界モデル検査ツール VARVEL

表 2.2　F-Soft/VARVEL が検出できる典型的なプログラミングエラー

検査の種類	検出できるエラー
ポインタ有効性	NULL/未初期化/解放済みポインタによる参照
配列境界	上限/下限違反 (添字とポインタによるアクセス)
文字列操作	バッファオーバーフロー/アンダーフロー
メモリ管理	メモリリーク

VARVEL ツールは，有界モデル検査法 [18] を用いて，逐次 C プログラムを検査する [67]．VARVEL は，F-Soft [54] の典型的なプログラミングエラーの検出機能 (表 2.2) に加えて，C 標準ライブラリ関数のスタブを提供し，さらに DbC によるモジュラー検証をサポートする．

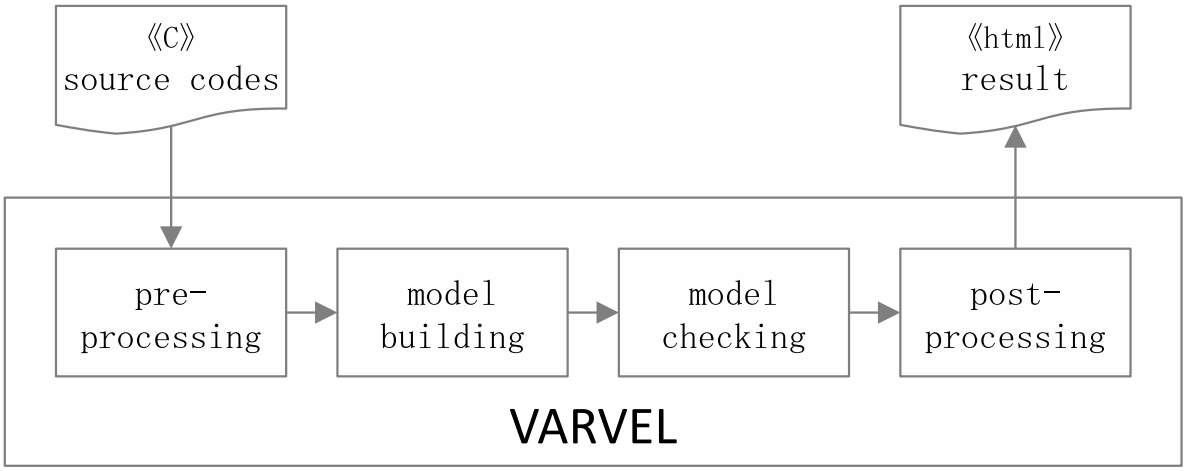


図 2.3　F-Soft/VARVEL の処理の流れ

VARVEL は，F-Soft の機能によって，C プログラムを論理式に変換して有界モデル検査を行う．図 2.3 に処理の流れを示す [45]．

まず，前処理では，入力として与えられた ANSI-C(C90) 準拠のソースコードを，C のサブセットの表現に直す．CIL [72] を用いて，次のような構文上の変換を行う．

- 関数ごとに，return 文を高々 1 回とする

- for 文や do 文を，これらの前後の箇所における変数値が同じとなる while 文にする

次に，モデル構築処理では，サブセットの表現に変換されたソースコードから制御フローグラフ (CFG. control flow graph) を構築し，CFG を表す論理式を生成する．有界モデル検査を実行できるように，CFG には大きく 3 つの変換 (変数に関する変換，制御に関する変換，最適化) を行う．

変数に関する変換． 有界モデル検査用の論理式に現れる変数は，真偽値を格納する命題変数である，一方，C 言語の変数には，計算機のメモリ概念に依存したポインタや，構造を持つ配列や構造体がある．C 言語における変数を，命題変数にマッピングする処理が必要となる．

- ポインタ型変数 `p` について，`p` 自身を表す変数 `p__ptr` と `p` を介して参照される値を示す変数 `p__deref` を導入する．
- 配列 `a` の `i` 番目の要素を変数 `a__i` とする．
- 構造体変数 `s` のフィールド `m` を変数 `s__flat__m` とする．(共用体も同様)

ポインタに代入され得るアドレスは，ポインタ解析 [77] によって求め，これらのアドレスを考慮して間接参照などを表現する．たとえば，ポインタ `p` にポインタ `pa` と `pb` のいずれかが代入されることが判ると，間接参照を用いた式 `x = *p` は次の式に変換される．

```
x =( p__ptr==pa__ptr ? pa__deref :
     ( p__ptr==pb__ptr ? pb__deref : x ) )
```

また，配列要素を用いた式 `x = a[i]` は，次の式に変換される．ここで配列 `a` のサイズを 2 とした．

```
x = ( i==0 ? a__0 : ( i==1 ? a__1 : x ) )
```

上式からわかるように，配列のサイズが大きいと，要素に対する変数の数が多くなり，状態爆発を起こす．さらに，リストのように無限の反復をもつ構造体は有界な表現に直せない．これらについては，利用者が指定した数までの配列要素や構造体フィールドを変数と

し，指定数を超える部分については，値の設定を無視し，値の取得では非決定的に選ばれた任意の値が取得されるものとする．たとえば，配列要素からの値の取得 a[i] は，次の式に変換される．ここで N は利用者が指定した数であり，α は非決定的に選ばれた任意の値である．

$$\texttt{a[i]} := \begin{cases} \texttt{a__i} & (\ \texttt{i}\in\texttt{0..N-1}\ ) \\ \alpha & (\ \texttt{N}\leq\texttt{i}\ ) \end{cases}$$

なお，VARVEL では，前記以外に，変数に関して次の制限がある．大域変数はプログラム開始時点で非決定的に選ばれた任意の値を取り得るものとする．ビットフィールドは各フィールドメンバを通常の構造体メンバとして扱い，ビット幅を考慮しない．共用体は構造体として扱う．構造体のポインタ型フィールドは，前記のリストと同様に扱い，利用者が指定した回数まではポインタによる参照を辿ることができる．指定回数を超える場合，辿った先のメンバは，値の設定を無視し，値の取得では非決定的に選ばれた任意の値が取得されるものとする．リストやツリーなどのポインタチェインは，同じポインタチェイン内の要素を参照しないものとして扱う．このため，ダブルリンクリストや循環リストは正しくは解析できない．volatile 修飾子は無視し，変数の値は明示的な代入によって変わるものとする．

制御に関する変換． VARVEL は，CBMC [25] と同様に，関数呼出しがあると，呼ばれる関数の本体コードを呼出し元のコード内にインライン展開する．ソースコードを与えていない関数は，大域変数に対する副作用を持たず，戻り値は非決定的に選ばれた任意の値を取り得る未解釈関数として扱う．コールツリーを調べて再帰呼出しを検知した場合は，一定の繰り返し回数だけインライン展開を行う．関数ポインタを介した関数呼出しについては，ポインタ解析 [77] によって関数ポインタにアドレスが代入される関数を特定し，インライン展開を行う．VARVEL に与えたソースコード内に，関数ポインタを介して呼ばれる関数が存在しない場合，関数ポインタを介した関数呼出しは，未解釈関数として扱う．インライン展開によってスケーラビリティは低下する．但し，VARVEL は 3.2 節の DbC の基本機能によるモジュラー検証をサポートすることで，スケーラビリティの問題を軽減する．

なお，VARVEL では，前記以外に，制御に関して次の制限がある．setjmp/longjmp による非局所分岐には対応しておらず，longjmp 関数を未解釈関数として扱う．また，signal 操作による割込みにも対応しておらず，検査対象の関数に関しては割込みが発生しないものとして検査を行う．

最適化．　モデル検査で探索する有限状態空間を小さくするために，変数や処理を減らす．定数畳込み (constant folding) では，変数の def-use 関係を辿り，変数に定数が代入された後で，別の代入なしにそのまま参照される場合に，その変数を定数で置き換える．たとえば，`x=定数;y=x;` を `y=定数;` とする．バックワードスライシング (backward slicing) では，プロパティに関連する変数についてのデータ・制御依存性をプロパティを検査する箇所からプログラムの開始箇所に向かって調べ，その変数に関係するプログラム断片を抽出し，無関係の箇所を除外する．たとえば，ポインタ有効性の検査の場合，ポインタ変数 `p` について参照外し (`*p`) を行う箇所からバックワードスライシングを行い，`p` に関係するプログラム断片を得る．

モデル構築処理では，最後に，前述の変換を行った CFG から有界モデル検査用の論理式 C とプロパティを表す論理式 P を生成する．C は変数ごとの値の遷移を表す式の論理積であり，各遷移式はプログラムのある位置から次の位置に制御が移る際に変数の値がどう変るか示す．P は，たとえば，NULL ポインタ参照が起きないというプロパティであれば，プログラム上の位置を指す組込み変数を pc，ポインタ p の参照外しの記述位置を l とすると，$(pc = l) \Rightarrow (p \neq NULL)$ となる．

続く，モデル検査処理では，$C \wedge \neg P$ を満たす変数値の割当があるかどうかを有界モデル検査により調べる．$C \wedge \neg P$ は，プログラムが実行されたときにプロパティが成り立つことを示す $C \Rightarrow P$ の否定である ($\Rightarrow$ は論理の含意)．変数値の割当が見つかれば，$C \wedge \neg P$ を示す証拠 (witness)，すなわち $C \Rightarrow P$ に対する反例 (counter-example) として出力する．見つからなければ，$C \wedge \neg P$ ではなく，その否定である $C \Rightarrow P$ が常に満たされるといえる．

最後に，後処理では，モデル検査の結果を，元のプログラム上の表現に置き換えて，利用者に可読な形式 (HTML) で出力する．

```
 5: void foo( int x, int y ){
 6:   x = x + y ;
 7:   if ( x != 1 )
 8:     x = 4 ;
 9:   else
10:     x++ ;
11:
12:   assert( x <= 3 ) ;
13: }
```

図 2.4　アサーション違反を起こすプログラムの例

図 2.4 のプログラム foo は引数 x と y の和が 1 以外の場合に，アサーション違反を起こす．VARVEL は図 2.5 のような検査結果を出力する．表の見出し行において，*Step*, *Line*, *Function*, *Detailed Info* は，それぞれ CFG 上の位置，ソースファイル上の行番号，関数名，そして変数への値の代入や成立した条件の状況などを表す詳細情報である．*Step* において 0 は初期状態を表す．詳細情報の $x\{-2\}$ は変数 x の値が -2 であることを示す．行番号のリンクをクリックすると，画面下側のソースコード表示で対応する行が青くハイライトされる．図 2.4 は，行番号 L12 のリンクをクリックしたところである．

VARVEL は，累積で 800 万行を超えるソースコードに適用されている．図 2.6 のコード断片に示すような，見つけにくい潜在的な不具合を指摘することに成功している．図 2.6 において，関数 func1 は func2 と func3 を呼び出す (それぞれ行 15,17)．func2 で malloc が失敗すると (行 3)，func3 で NULL ポインタ参照が起きる (行 9)．malloc の失敗のような欠陥は発見するのが難しい．実行時の環境の振る舞いに依存するため，テストで発見しようとすると，大変多くのテストケースを必要とすることになる．

VARVEL Witness _fSfT_du

Witness shows: Assertion Violation (x <= 3)

Step Information

normal mode maintenance mode

Step	Line	Function	Detailed Info
0	L5	foo	y =-2
0	L5	foo	x =-2
5	L5, L5	foo	y {-2} = y {-2}
5	L5, L5	foo	x {-2} = x {-2}
10	L6	foo	x {-4} = (x {-2} + y {-2})
12	L7	foo	*成立した条件* : x {-4} != 1
13	L8	foo	x {4} = 4
15	L12	foo	*成立した条件* : x {4} > 3
16		foo	**Witness shows:** Assertion Violation (x <= 3)

```
00005 void foo( int x, int y ){
00006   x = x + y ;
00007   if ( x != 1 )
00008     x = 4 ;
00009   else
00010     x++ ;
00011
00012   assert( x <= 3 ) ;
00013 }
```

図 2.5　VARVEL の検査結果の画面例

```
 1: int func2( char**bufp, char* path ){
 2:   int len = 0 ;
 3:   *bufp = malloc( strlen( path ) ) ;
 4:   if( *bufp ) len = strlen( *bufp ) ;
 5:   return  len ;
 6: }
 7:
 8: int func3( const char *buf ){
 9:   char c = *buf ;
10:   return c ;
11: }
12:
13: int func1( char *path ){
14:   char *buf ;
15:   int len = func2( &buf, path );
16:   /* ... about 150 lines *** */
17:   if ( func3( buf ) ) return -1 ;
18:   return 0 ;
19: }
```

図 2.6　NULL ポインタ参照を起こすプログラムの例

第 3 章

VARVEL におけるモジュラー検証

3.1 背景

ソフトウェアの品質保証では，プログラム・テスト技術が大きな役割を担っている．近年，システム障害の社会問題化により，ソフトウェアの更なる品質向上に関心が高まっている．テスト技術を補う品質保証手段が必要である．

従来のテスト技術では，入力データを対象プログラムに与えて実行し，その出力を期待結果と比較して，プログラムの正しさを確認する．モジュラー検証では，開発者が事前・事後条件といった仕様をプログラムに記述し，検査ツールが事前条件を前提として対象プログラムをモジュールごとに解析し，プログラムの事後状態において事後条件が成立することを調べる．モジュラー検証の事前条件と事後条件は，それぞれ従来テスト技術における入力データと期待結果に相当するので，テスト作業を自動化する手段として期待されている．

モジュラー検証ツールでは，経路を考慮しない (path insensitive) 定理証明を用いることが多く，誤警告が多いという問題がある [30] ．本章では，経路を考慮する (path sensitive) 有界モデル検査を用いたモジュラー検証を提案する．提案方法における記法と実現方式を説明し，実験を通じて単体テストの自動化につながる結果が得られたことを述べる．

3.2 実現方式

VARVEL の基盤である F-Soft は，CBMC [25] や Bogor [75] と同様に，*assert* と *assume* という 2 つのプリミティブを提供する．VARVEL では，利用者はプリミティブを関数の形式（`__assert` と `__assume`）で用いる．

プリミティブに与える条件を C，プリミティブを評価する前の状態を S とすると，$assert(C)$ は $S \wedge C$ が成り立つかどうかを検査すべきことを示す．また，$assume(C)$ は評価後の状態を $S \wedge C$ とすべきことを示す．利用者が，これらのプリミティブを用いて，ソースコードを補う情報をツールに与えることにより，近似による誤警告を削減できる．しかし，プリミティブの使い方は明確には決まっておらず，実際の作業では検査効率を向上させる系統的な使い方を示す必要がある．

表 3.1　VARVEL の DbC 記法

仕様の種別	説明
@invariant C	論理式 C は大域データの不変条件である
@pre C	論理式 C は関数の事前条件である
@post C	論理式 C は関数の事後条件である
@param[out] v	変数 v は関数によって更新される

組込み変数・関数	説明
__return	関数の戻り値
__length(a)	配列 a の要素数を返す
__strlen(s)	文字列 s の文字数を返す
__old(v)	関数開始時の変数 v の値を返す

VARVEL は，DbC を明示的にサポートするために，DbC の記法 (表 3.1) を提供する．事前条件などの論理式では，構造体メンバへのアクセス，ポインタの参照外しなど，C 言語の式で用いる表現を用いることができる．DbC のサポートは，F-Soft にはない，VARVEL 独自の機能である．

VARVEL は，モジュラー検証を実現するために，C プログラムのコメントに記述された DbC 仕様を，先のプリミティブを用いたプログラムコードに変換して，有界モデル検査を行う．例えば，検証対象の関数については，その事前条件 Pre を`__assume(Pre)`

に，事後条件 *Post* を`__assert(Post)` に変換して，それぞれ関数本体の前後に挿入する [44] .

DbC 仕様からプリミティブを用いたプログラムコードへの変換方法は，検査対象である関数の使われ方によって異なる.

- **エントリ関数**：利用者がソースコードを与え，かつコールグラフの起点として指定した関数
- **非エントリ関数**：利用者がソースコードを与え，かつコールグラフの起点として指定しなかった関数
- **ライブラリ関数**：利用者がソースコードを与えなかった関数（かつ，エントリ・非エントリ関数から呼ばれる関数）

図 3.1 と図 3.2 には，それぞれ関数の使われ方に応じた，DbC 仕様からプログラムコードへの変換方法と変換例を示す.

変換前

```
/**
  @pre P
  @param[out] v
  @post Q
*/
func(...){
  Body
}
```

変換後

(エントリ関数)

```
func(...){
  __assume( P );
  Body
  __assert( Q );
}
```

(非エントリ関数)

```
func(...){
  __assert( P );
  __assume( P );
  Body
  __assert( Q );
  __assume( Q );
}
```

(ライブラリ関数)

```
func(...){
  __assert( P );
  __assume( P );
  v=__NONDET__();
  __assume( Q );
}
```

図 3.1　DbC 仕様のプログラムコードへの変換方法
__NONDET__は未解釈関数 (uninterpreted function) であり，非決定的に選ばれた任意の値を返し，副作用を持たない.

エントリ関数の場合，VARVEL は，事前条件と不変条件をそれぞれ関数の開始箇所における *assume* プリミティブに変換する．また，事後条件と不変条件をそれぞれ関数の終了箇所における *assert* プリミティブに変換する．これにより，VARVEL は，事前条件の成立を前提として，エントリ関数のボディを探索し，関数の終了箇所において事後条件が成立することを検査する．VARVEL は，エントリ関数の引数や大域変数など対象関数の

対象プログラム

```
unsigned int g = 0;
/**
    @invariant  g <= 1
*/

/**
    @pre   0 < x
    @param[out]  g
    @post  0 <= __return
    @post  g == __old(g) - 1
*/
int foo( int x ){
    int result ;
    result = x - (g--) ;
    return  result ;
}
```

エントリ関数の場合

```
int foo( int x ){
    int __old_g = g ;
    int result ;
    __assume( 0 < x ) ;
    __assume( g <= 1 ) ;
    result = x - (g--) ;
    __assert( 0 <= result ) ;
    __assert( g == __old_g - 1 ) ;
    __assert( g <= 1 ) ;
    return  result ;
}
```

非エントリ関数の場合

```
int foo( int x ){
    int __old_g = g ;
    int result ;
    __assert( 0 < x ) ;
    __assume( 0 < x ) ;
    __assert( g <= 1 ) ;
    __assume( g <= 1 ) ;
    result = x - (g--) ;
    __assert( 0 <= result ) ;
    __assume( 0 <= result ) ;
    __assert( g == __old_g - 1 ) ;
    __assume( g == __old_g - 1 ) ;
    __assert( g <= 1 ) ;
    __assume( g <= 1 ) ;
    return  result ;
}
```

ライブラリ関数の場合

```
int foo( int x ){
    int __old_g = g ;
    int result ;
    __assert( 0 < x ) ;
    __assume( 0 < x ) ;
    __assert( g <= 1 ) ;
    __assume( g <= 1 ) ;
    result = __NONDET__() ;
    g = __NONDET__() ;
    __assume( 0 <= result ) ;
    __assume( g == __old_g - 1 ) ;
    __assume( g <= 1 ) ;
    return  result ;
}
```

図 3.2　DbC 仕様のプログラムコードへの変換の例

外部で値が決まる変数は，エントリ関数の開始箇所において非決定的に選ばれた任意値を取りうることを，暗黙の前提とする．事前条件あるいは不変条件についての *assume* は，暗黙の前提を上書きする．

非エントリ関数の場合，VARVEL は，事前条件と不変条件をそれぞれ関数の開始箇所における *assert* と *assume* プリミティブに変換する．同様に，事後条件と不変条件もそれぞれ関数の終了箇所における 2 種類のプリミティブに変換する．この変換により，VARVEL は，呼出し元については，呼ばれる側の事前条件を守って関数呼出し行っていることを検査し，呼ばれる側の事後条件の成立を前提として，呼び出し後のコードを探索する．呼ばれる側については，その事前条件の成立を前提として，そのボディを探索し，ボディの終了箇所においてその事後条件が成立することを検査する．

ライブラリ関数の場合，その DbC 仕様はプロトタイプ宣言に付記されて，VARVEL に与えられる．VARVEL はライブラリ関数のスタブ関数を作成し，事前条件と不変条件をそれぞれスタブ関数の開始箇所における *assert* と *assume* プリミティブに変換する．また，事後条件と不変条件をそれぞれ関数の終了箇所における *assume* プリミティブに変換する．戻り値や DbC 仕様において関数によって変更されると明示された変数については，未解釈関数を用いて非決定的に選ばれた任意値を代入するコードを生成し，事後条件の前に置く．図 3.2 において，大域変数 `g` の値が関数内で変更されることが，DbC 仕様（`@param[out] g`）によって明示的に指定されている．戻り値は，関数内で値が変更されるものと，暗黙的に見なされる．この変換によって得たスタブコードを用いて，VARVEL は，ライブラリ関数の呼び出しにおいて，その事前条件を守っていることを検査し，その事後条件の成立を前提として，呼び出し後のコードを探索する．

これらの一連の変換により，VARVEL は次の式を満たす変数値の割当があるかどうかを有界モデル検査により調べることができる．

$$(Pre \wedge C) \wedge \neg Post$$

変数値の割当があれば，それは事前条件を前提としたプログラム実行で起こりうる事後条件違反を示す反例である．検証者が DbC 記法に則って関数の機能仕様を記述することで，VARVEL ではプリミティブを系統的に使うことができる．

3.3 実験

DbC の基本機能 (節 3.2) は，VARVEL の新機能である．その基本的な実験結果を表 3.2 に示す．実験に用いたプログラム *prog1* と *prog2* は，それぞれ同じシステム管理ソフトウェア製品の一部であり，単体テスト前のものである．各プログラムの開発者が DbC 仕様を記述した．VARVEL による検査と並行して，単体テストが行われた．表 3.2 の列において，*Size*，*File*, *Func*, *DbC*，*Man-hr*, *Bugs* は，それぞれプログラムの行数 (KLoc)，ファイル数，関数の数，開発者が記述した DbC 仕様の行数，DbC 仕様の記述に掛かった工数 (人時)，そして，修正すべき実際のバグであると開発者によって判断された DbC 仕様違反の数である．

表 3.2　DbC 基本機能の実験結果

Program	Size	File	Func	DbC	Man-hr	Bug
prog1	1.0	2	30	68	3	1
prog2	5.6	8	78	260	16	3

単体テストでは，プログラム *prog1* と *prog2* に関して，それぞれ 2 個と 3 個のバグが見つかった．*prog1* では，単体テストで見つかった 2 個のバグが，VARVEL の DbC 基本機能によって警告されると期待したが，1 個のバグしか警告されなかった．バグを含む関数が防衛的プログラミングのスタイルで記述されていたことが，警告できなかった理由である．開発者は引数に許される値を DbC の事前条件に記述し，引数をチェックして，その値が許されない場合にはエラー処理を行うプログラムコードを記述した．VARVEL は，開発者が記述した DbC 仕様のもとでは，引数が許されない値である場合のエラー処理をデッドコードであると見なし，バグのある箇所を探索しなかった．防衛的プログラミングに起因するバグの見逃ごしは，検査ツール自体で防ぐことはできない．引数チェックのエラー処理も含めて，関数の振舞いを DNF 形式で明確に分けて書くように，開発者を教育する必要がある．別の問題として，VARVEL は動的にメモリにロードされる関数に関して誤警告を発した．この関数は関数ポインタを介して間接的に呼び出されており，関数についてのソースコードは宣言すら無いことと，開発者は関数ポインタ関して DbC 仕

様を記述できないことが，誤警告の理由であった．この誤警告の問題を契機となり，第 5 章の関数ポインタ用の DbC 記法と検査方法を提案することとなった．

図 3.3 は，プログラム *prog2* で発見されたバグの例である．事後条件は，関数 `GetNList` の戻り値と引数`*pErr` が同時に 0 になることを述べている (行 4)．しかし，`GetNList` 内で呼び出される `GetHost` が 0 を返すと (行 13)，`*pErr` は `0x02` となり (行 15)，後続のコードにおいて戻り値だけが 0 となる (行 20)．このため，関数の終了箇所 (行 24) において，事後条件違反が検出される．

```
 1: /**
 2:  @pre  flag == 1 || flag == 2
 3:  @pre  pList != NULL && pErr != NULL
 4:  @post __return == 0 && *pErr == 0  || \
 5:        __return == -1 && \
 6:          *pErr > 0 && *pErr < 0x40
 7: */
 8: int GetNList( int flag, NList
 9:               *pList, int *pErr ) {
10:   int err, ret ;
11:   char *name ;
12:   /* ... */
13:   ret = GetHost( name, &err );
14:   if(! ret){
15:     *pErr = 0x02 ; goto ErrExit ;
16:   }
17:   goto Exit ;
18:   /* ... several 10s of lines ... */
19: ErrExit:
20:   ret = Cleanup( &err );
21:   if (! ret) *pErr = err ;
22: Exit:
23:   return ret;
24: }
```

図 3.3　事後条件違反を起こすプログラム例

本実験では，ソースコードは開発中であったため，開発者が手作業で DbC 仕様を記述した．しかし，DbC 仕様の条件の注釈は 1 行当たり数分で済んだ．また，記述コストに

ついて開発者から否定的なコメントはなかった．それよりも，開発者からは次のような肯定的なフィードバックを得た．

- DbC 仕様の注釈を通して，プログラムの仕様を見直す機会を得た．
- 自動解析による早期バグ発見ができた．

3.4 考察

本章では，有界モデル検査ツール VARVEL における DbC 仕様の記法と，DbC 仕様から BMC の検証プリミティブ関数への変換によるモジュラー検証の実現方法を述べた．また，産業界の C プログラムを対象とした実験では，単体テストにおいてテスト仕様書を書くのと同様に，開発者自身が DbC 仕様を記述し，VARVEL で検査を行い，モデル検査の反例についてプログラムを調べてバグを発見できること，そのバグは実験とは独立に行った単体テストで発見したバグと同じであることを示した．テスト作業の自動化に，ソフトウェアモデル検査が利用できることを示す結果が得られた．

なお，本章の実験では，誤警告によって無駄な不具合の原因調査を行う問題も判った．誤警告は，対象関数から呼ばれる関数に DbC 仕様が付記されていないことや，大域的な情報の不足に起因する．第 4 章と第 5 章で，これらの問題についての解決策を示す．

3.5 関連研究

一般に，DbC 仕様の記述コストは，DbC 導入の阻害要因と見なされている．この問題を解決するために，Hackett ら [43] は，十分にテストされた既存ソースコードからの自動注釈推論を提案し，SALInfer ツールとして実装した．SALInfer は，ポインタ型変数が NULL か否か，配列のサイズはいくつかといった実行時エラーを回避するための仕様を推論する．

しかし，関数の振舞い・機能仕様は，プログラムコードから自動推論することは難しく，人手で記述する必要がある．ID カード用 Java アプレットを対象としたモジュラー検証の事例 [73] では，プログラムコード 3 行に対して DbC 仕様 1 行を記述した．このよう

に，品質向上を目的とするのであれば，プログラムの正しさの基準となる仕様を検査ツールに与えることは当然である．本研究も，開発者は自らが設計あるいは実装する関数についてプログラム単体テストにおける入力データや期待結果に相当する DbC 仕様を書くことが望ましいと考える．

第 4 章

モジュール再入に関する検査

4.1 背景

モジュラー検証は，プログラム全体の検査を，モジュールごとの検査に分けて行う考え方である．信頼できる検査結果を得るには，そのモジュールを呼び出すプログラムやそのモジュールから呼ばれるプログラムといった外部環境を補う情報を検査ツールに与える必要がある．モジュール外部の情報は，事前・事後条件といった DbC 仕様として与えられる．事前条件を前提として，モジュールを実行し，実行後の状態が事後条件を満たすことを検査する．対象モジュールが他のモジュールを呼び出す場合，呼ばれる側のモジュール本体の代わりに，その DbC 仕様を用いて，プログラムを解析する．ソフトウェアモデル検査にモジュラー検証の考え方を取り入れることにより，モデル検査のスケーラビリティの問題を軽減できる．

しかし，モジュラー検証では，手続きごとの情報しか調べないので，コールシーケンスに関わる大域的な情報が不足し，誤警告 (false alarm) が発生することが知られている [35] [65] ．産業界で普及している C プログラムでは，関数間での大域変数を用いたデータの受け渡しが広く行われている．プログラムを静的に検査する上で，大域変数に因る誤警告を減らす工夫が必要である．大域変数によるデータの受け渡しでは，様々な関数が大域変数をどう更新するかを把握することが難しく，誤警告が起こり易い．モジュラー検証では，大域変数の値を不変条件によって制限することにより，手続きごとの検査の精度を上げることができる．有界モデル検査を用いたソフトウェアモデル検査では，コールシー

ケンスに関わる関数ボディをインライン展開して，大域変数の値の設定と取得箇所をひとまとめに検査することにより，スケーラビリティは下がるが，正確な検査を行える．しかし，再入の状況下では，いずれの方法を用いても，静的な検査が困難なことがある [35] [65] .

本章では，再入の問題に対して，不変条件を検査すべき範囲 (scope) を指定する DbC 記法と，再入を起こすプログラムの検査方法を提案する．また，例題プログラムを用いた実験により，提案方法によって適切な箇所で不変条件違反を検出できることを示す.

4.2 問題

本章では，あるモジュールの関数が実行中に，そのモジュールの関数が呼び出されることを「モジュールへの再入」あるいは単に「再入」(re-entrancy) と呼ぶ.

```
class Subject {
  Observer obs; int state;
  invariant state >= 0;
  void Update(int i) {
1:  this.state = i;
2:  obs.Notify();
    if (this.state < 0) {
      state = 0;
    }
  }
  int Get() {
1:  return this.state;
  }
}
```

```
class Observer {
  Subject sub; int cache;
  void Notify() {
1:  this.cache = sub.Get();
  }
}

void testObserver() {
1: Observer o = new Observer();
2: Subject s = new Subject();
3: s.obs = o;
4: o.sub = s;
5: s.Update(-10);
}
```

図 4.1　オブジェクト再入の例 (C#)

文献 [35] の Fig. 1. Subject-Observer sample を転載.

文献 [35] におけるオブジェクト指向言語 C#での再入の例を図 4.1 に示す．ただし，関数 `testObserver` がメソッド `Update` で与える引数は文献 [35] での 10 から再入時に不変条件違反を起こす-10 に変更した．関数 `testObserver` から始まる下記のコールシーケンスにおいて，メソッド `Get` が呼び出された時点で，クラス `Subject` のオブジェクト `s`

への再入が起きる．

クラス			Subject		Observer		Subject
オブジェクト			s		o		s
メソッドなど	testObserver	$\longrightarrow$	Update	$\longrightarrow$	Notify	$\longrightarrow$	Get

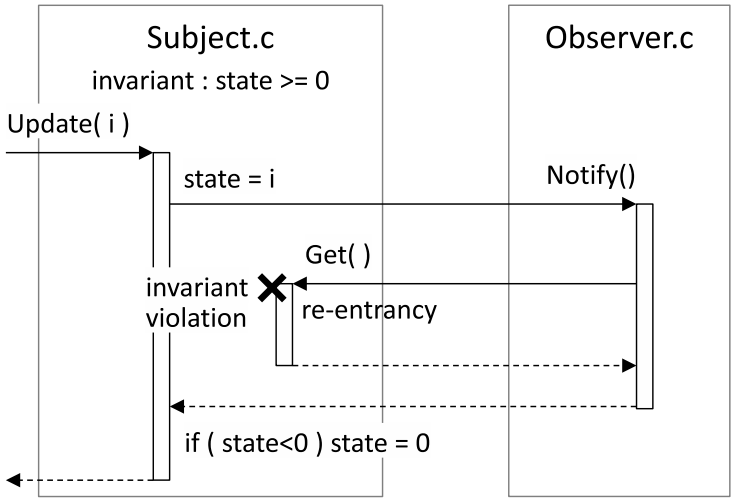
クラス `Subject` にはクラス不変表明 `state >= 0` が定義されている．`Subject` のメソッド `Update` は，まず引数 `i` をフィールド `state` に代入する．関数 `testObserver` は `Update` の引数 `i` に 0 未満の値 (`-10`) を渡しているので，この代入の時点で不変条件は成立しなくなる．次に，`Update` はクラス `Observer` のメソッド `Notify` を呼び出し，`Notify` は `Subject` のメソッド `Get` を呼び返す．`Get` の開始時点，すなわちオブジェクトへの再入の時点で，不変条件に違反する．

メソッドごとにモジュラー検証を行うと，不変条件に違反しない事前状態からメソッド `Get` のボディを実行し，その事後状態が再び不変条件に違反しないことを検査するため，前記の不変条件違反を見逃ごす．対象プログラムが不変条件の成立を前提とするならばモジュラー検証では不変条件違反を検出できず，不変条件の成立を前提としないのであればモジュラー検証は検査の仕方として意味がない．

手続き型言語 C では，さらに問題が増える．図 4.2 は，図 4.1 のソースコードを C 言語で記述した例である．ファイル `Subject.c` と `Observer.c` がそれぞれ論理モジュール *Subject* と *Observer* であるとすると，関数 `testObserver` から始まる下記のコールシーケンスにおいて，関数 `Get` が呼び出された時点で，*Subject* への再入が起きる．

ファイル			Subject.c		Observer.c		Subject.c
関数	testObserver	$\longrightarrow$	Update	$\longrightarrow$	Notify	$\longrightarrow$	Get

関数 `testObserver` は実引数として`-10` を与えて，モジュール *Subject* の関数 `Update` を呼び出す (図 4.2 ラベル t1)．`Update` は，まず引数 `i`(値は`-10`) を大域変数 `state` に代入する (図 4.2 ラベル s1)．`state` の値は 0 未満 (`-10`) なので，不変条件 `state >= 0` は成立しない．次に，`Update` はモジュール *Observer* の関数 `Notify` を呼び出す (図 4.2 ラベル s2)．`Notify` はモジュール *Subject* の関数 `Get` を呼び返す (図 4.2 ラベル o1)．ファ



Subject.c

```
int state = 0 ;
/**
@invariant state >= 0
*/

void Update(int i) {
s1: state = i ;
s2: Notify() ;
    if (state < 0) {
        state = 0 ;
    }
}

int Get() {
    int r ;
s3: r = state ;
    return  r ;
}
```

Observer.c

```
int cache ;
void Notify() {
o1: cache = Get() ;
}
```

Test.c

```
void testObserver() {
t1: Update(-10) ;
}
```

図 4.2　モジュール再入の例 (C)

イル `Subject.c` の不変条件がモジュール $Subject$ のみの制約であるとすると，`Get` の開始時点で不変条件違反が起こる．

図 4.1 の例に用いた C#のようなオブジェクト指向言語では，クラスの内部に記述された不変条件はクラス不変条件である．オブジェクト指向言語におけるクラス不変条件のスコープは，不変条件が定義されたクラスの各インスタンスであり，これらのインスタンスにおけるメソッドの開始箇所と終了箇所でクラス不変条件を検査する．クラス不変条件は他クラスのインスタンスのメソッド呼び出し時には検査しない．一方，C は手続き型言語であり，オブジェクト指向言語におけるクラスのような，変数や関数をまとめる言語要素を持たない．C 言語における不変条件のスコープは通常はプログラム全体であり，全ての関数の開始および終了時に不変条件が成立しなければならない．

ファイル `Subject.c` に関してモジュラー検証を行うと，検査ツールは関数 `Update` と `Get` を別々に検査する．関数 `testObserver` から始まるコールシーケンスのコンテキストが与えられていないため，ツールは前述の不変条件違反を発見できない．代わりに，ツールは関数 `Notify` の呼び出し箇所で，不変条件違反を警告する (図 4.3)．これは，ツールが不変条件 `state >= 0` のスコープを大域と見なし，全ての関数の開始および終了時に不変条件を評価するためである．コールシーケンスのコンテキストを考慮すると関数 `Get` の開始時点で不変条件違反を検出したいが，モジュラー検証では `Get` の開始時点での不変条件違反を見逃ごし，関数 `Notify` の開始時点で不変条件違反を誤検出する．

4.3　検査の方法

各論理モジュールがそれぞれ異なるファイルに実装されるとした場合に，各モジュールをスコープとする不変条件への違反を再入が起きた箇所で発見したい．このような再入の状況下で，C プログラムの不変条件に対する検査を行うためには，次の機能が必要である．

- ファイルをスコープとする不変条件
- 関数の境界を越えた情報の利用

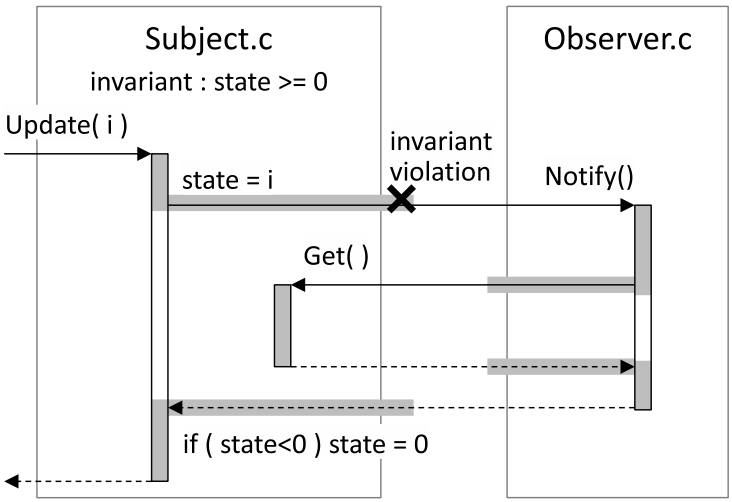


図 4.3　モジュラー検証による再入時の不変条件の検査

関数 Get の開始時点で検出したい不変条件違反が見逃ごされ，代わりに関数 Notify の呼び出しの際に不変条件違反が誤って検出される

ファイルをスコープとする不変条件．　ある論理モジュール内でのみ検査が必要な不変条件に関して，静的検査ツールに与えた対象プログラムに関連する全ての関数の開始および終了時に不変条件を検査する方法では，他モジュールの関数の開始時点において不変条件違反を検出してしまい，再入箇所における不変条件違反を見逃ごす．静的検査ツールが，再入が起きる以前の箇所で不変条件違反を検出してしまい，その違反が再入箇所における不変条件違反を隠してしまうためである．不変条件違反の見逃ごしと誤検出の 2 つの問題が生じている．これらの問題を解決するために，まず不変条件のスコープを特定の範囲に制限する必要がある．C 言語における一つのやり方として，スコープをファイル単位とすればよい．図 4.2 の例では，ファイル `Subject.c` で定義された不変条件 `state >= 0` のスコープは，`Subject.c` 自体に制限すべきである．スコープがファイルであるような不変条件を，**ファイル不変条件** (file-scoped invariant) と呼ぶこととする．図 4.4 に，ファイル不変条件の例 (`@file_invariant state >= 0`) と，ファイル不変条件を (アサーションなどの) プリミティブを用いて実装する例を示す．図 4.4 の実装は，図 3.2 の「エントリ関数の場合」に対応しており，関数 `Update` のみ，`Get` のみをモジュラー検証するための実装である．モジュラー検証を行うと，関数 `Update` が `Notify` を呼び出す際の不変条件の誤検出は発生しなくなる．しかし，コールシーケンスのコンテ

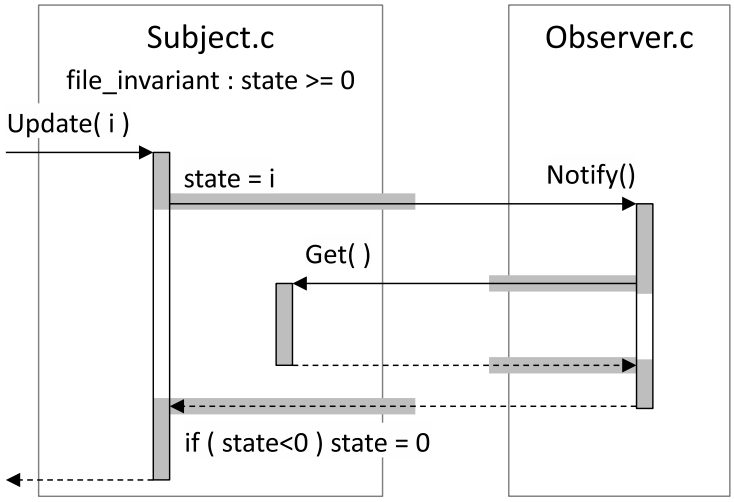
キストが無いことから，`Get` の開始時点で起こる不変条件違反を見逃ごす問題は残る．

関数の境界を越えた情報の利用．　F-Soft を含む，SAT に基づく有界モデル検査ツールは，与えられた全ての関数を 1 つの大きな関数にインライン展開する．この方法により，有界モデル検査ツールは，モジュラー検証において関数の境界となる DbC 仕様を超えて，呼ばれる側のソースコードの情報を使うことができる．VARVEL でも，F-Soft によって提供されるこの方法を利用できる．インライン展開の機能を用いて，検査対象モジュールを各関数から

`testObsesrver` $\longrightarrow$ `Update` $\longrightarrow$ `Notify` $\longrightarrow$ `Get`

のコールシーケンスに拡張した例を図 4.5 に示す．

図 4.5 の実装は，図 3.2 の「非エントリ関数の場合」に対応しており，関数 `Update` と `Get` がそれぞれ関数 `testObsesrver` と `Notify` から呼びされることを考慮した実装である．



Subject.c with
file-scoped invariant

```
int  state = 0 ;
/**
  @file_invariant  state >= 0
*/
void Update(int i) {
s1: state = i ;
s2: Notify() ;
    if (state < 0) {
        state = 0 ;
    }
}
int Get() {
    int r ;
s3: r = state ;
    return  r ;
}
```

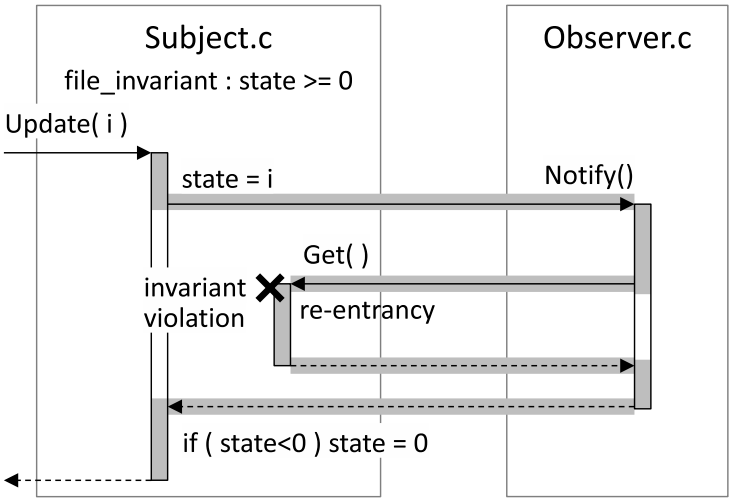
Subject.c with instrumented primitives
for *file-scoped invariant*

```
int  state = 0 ;

void Update(int i) {
    __assume( state >= 0 ) ;
s1: state = i ;
s2: /* Notify() ; */// No side-effects */
    if (state < 0) {
        state = 0 ;
    }
    __assert( state >= 0 ) ;
}
int Get() {
    int  r ;
    __assume( state >= 0 ) ;
s3: r = state ;
    __assert( state >= 0 ) ;
    return  r ;
}
```

図 4.4　ファイル不変条件の例

ファイル Observer.c については，元のソースコードからの変更が生じないため省略する．ファイル Subject.c における不変条件のスコープの変更により，ファイル Observer.c の関数 Notify には不変条件を検査するためのプリミティブを挿入する必要がなくなった．



Subject.c with
file-scoped invariant

```
int  state = 0 ;
/**
  @file_invariant  state >= 0
*/
void Update(int i) {
s1: state = i ;
s2: Notify() ;
    if (state < 0) {
        state = 0 ;
    }
}
int Get() {
    int r ;
s3: r = state ;
    return  r ;
}
```

Subject.c with instrumented primitives
for *file-scoped invariant*

```
int  state = 0 ;

void Update(int i) {
    __assume( state >= 0 ) ;
s1: state = i ;
s2: Notify() ;
    if (state < 0) {
        state = 0 ;
    }
    __assert( state >= 0 ) ;
}
int Get() {
    int  r ;
    __assert( state >= 0 ) ;
    __assume( state >= 0 ) ;
s3: r = state ;
    __assert( state >= 0 ) ;
    __assume( state >= 0 ) ;
    return  r ;
}
```

図 4.5　モジュールを拡大した検査の例

図 4.4 と同様に，ファイル Observer.c については，元のソースコードからの変更が生じないため省略する．

4.4 実験

図 4.2 の例を用いて，関数を検査対象モジュールとする検証，コールシーケンスを検査対象モジュールとする検証，ファイル不変条件を用いてコールシーケンスを検査対象モジュールとする検証の 3 つの場合について，モジュール（同図の `Subject.c`）の不変条件（同図の `state >= 0`）への違反を，モジュール再入箇所（同図の関数 `Get` の開始箇所）で検出できるか否かを実験によって確認した．確認結果を，表 4.1 に示す．

表 4.1　モジュール再入時の不変条件違反の検出

検証方法	検査結果	違反の検出箇所
モジュール=関数	違反なし	
モジュール=コールシーケンス	違反あり	関数 `Notify`
モジュール=コールシーケンス + ファイル不変条件	違反あり	関数 `Get`

関数をモジュールとする検証では，不変条件違反を検出しなかった．

コールシーケンスをモジュールとする検証では，図 4.6 に示すように，関数 `Notify` の開始箇所で，不変条件違反を検出した．モジュール再入箇所とは異なる箇所での検出であり，誤警告である．

ファイル不変条件を用いたコールシーケンスをモジュールとする検証は，図 4.5 の実装例を用いて行った．図 4.7 に示すように，関数 `Get` の開始箇所で，不変条件違反を検出した．期待どおりにモジュール再入箇所でファイル不変条件への違反を発見できた．

VARVEL Witness fsoft_fl.1.0.

Witness shows: Assertion Violation (state >= 0)

Step Information

normal mode maintenance mode

Step	Line	Function	Detailed Info
0	L6	Update	i =-1
8	L6, L6	Update	i {-1} = i {-1}
16	L7	Update	state {-1} = i {-1}
22	L2	Notify	*成立した条件* : state {-1} < 0
23		Notify	**Witness shows:** Assertion Violation (state >= 0)

図 4.6　モジュール再入におけるコールシーケンスを検証単位とする検証結果

VARVEL Witness fsoft_fl.1.0.

Witness shows: Assertion Violation (state >= 0)

Step Information

normal mode maintenance mode

Step	Line	Function	Detailed Info
0	L10	Update	i =-1
5	L10, L10	Update	i {-1} = i {-1}
13	L11	Update	state {-1} = i {-1}
23	L18	Get	*成立した条件* : state {-1} < 0
24		Get	**Witness shows:** Assertion Violation (state >= 0)

図 4.7　モジュール再入におけるファイル不変条件を用いたコールシーケンスを検証単位とする検証結果

4.5 考察

モジュラー検証では，大域的な情報の不足により，不具合の見逃ごしや誤警告を生じることがある．大域変数の値を制約する不変条件を定義する，あるいは，SAT に基づく有界モデル検査ツールのもつインライン展開の機能を用いてモジュールを関数からコールシーケンスに拡大することにより，大域的な情報を補うことができる．しかし，あるモジュールが関数を実行している最中に，別のモジュールから当該モジュールの関数の実行が要求されるという再入の状況においては，前述の対処だけでは，C プログラムにおける再入箇所における不変条件違反を検出することはできない．

4.3 節では，C プログラムにおける不変条件のスコープを大域からファイルに制限したファイル不変条件の記法の導入と，有界モデル検査ツールのインライン展開機能を用いた関数からコールシーケンスへの検証対象モジュールの拡大によって，それぞれ不変条件違反の誤警告と見逃ごしを排除する方法を提案した．また，有界モデル検査ツール VARVEL を用いた実験により，提案方法によって，ファイル不変条件への違反を再入箇所で検出できることを確認した．

本章の記法・検査方法は，大域的な情報の不足により不具合の見逃ごしや誤警告を生じるモジュラー検証の欠点を解消するものであるが，検査ツールの実用化に向けては不変条件に関して他の側面も考慮する必要がある．以下，順に述べる．

初期化処理．　不変条件は初期化が完了した後に成立する．C++ や Java のようなオブジェクト指向言語には，初期化のための言語要素であるコンストラクタがある．しかし，手続き型言語である C には，コンストラクタは無い．また，変数宣言の初期化子には，複雑なデータ構造を初期化するのに十分な表現力がない．C プログラマは，自分で初期化関数を書くことが多い．静的検査ツールに初期化関数を指示するような DbC 記法の拡張が必要である．具体的には，下記 2 種類の記法が必要である．

- 関数の開始時に不変条件を評価しない．終了時には評価する．
- 関数の開始および終了時に不変条件を評価しない

前者は，関数がオブジェクト指向言語のコンストラクタに相当する（初期化関数である）ことを指定する記法である．指定された関数は初期化全体を担い，その処理の終了後は，不変条件が成立していると期待される．後者は，関数が初期化関数から呼び出されることを指定する記法である．指定された関数は初期化の一部を担い，その処理の終了後も，別の初期化処理が行われる可能性がある．4.6 節に後述する JML[60] の helper プラグマと同様の役割を持つ記法である．

ヘッダファイルの不変条件のスコープ．　4.3 節では，ソースファイル (ファイル拡張子を".c"とする) に関して，不変条件のスコープを説明した．ヘッダファイル (ファイル拡張子を".h"とする) の場合，ファイルのスコープに少なくとも 2 つの解釈があり得る．一方の解釈では，ヘッダファイルに定義された不変条件の検査を，同じヘッダファイルで宣言された関数の開始と終了時に行う．他方の解釈では，ヘッダファイルに定義された不変条件の検査を，そのヘッダファイルを参照するソースファイルで定義された関数の開始と終了時に行う．プログラムの論理モジュールと物理的なコンテナ (ファイル，ディレクトリ，実行モジュール) との関係が明確にできると，これらの解釈について良い選択を行えるであろう．

論理モジュール．　4.3 節では，1 つのソースファイルを 1 つのモジュールと見なした．しかし，複数のソースファイルを論理的には 1 つのモジュールと見なして，プログラムを書くことは多い．ソースファイルがどの論理モジュールに属するかを明示する次のような記法があるとよい（4.3 節の@file_invariant は，この記法の特殊な場合といえる）．

@module M	ファイルは論理モジュール M に属する
@module_invariant(M) Inv	不変条件 Inv のスコープは論理モジュール M である

C 言語の構文や意味だけを考えるのでは，これらの課題の解決は容易ではない．開発者が産業界のソースコードをどのように書いているか（初期化関数など），開発者が言語要素や物理コンテナを用いてどのように論理モジュールを決めているか，といった現場で実際に行われているプログラミングの慣習をよく理解して，モジュールの概念を定義しなければならない．

4.6 関連研究

オブジェクト指向プログラミング言語に関しては，オブジェクト再入時に不変条件をどう検査すべきか，様々な議論がなされている．

Meyer は，機能を要求する *client* と機能を提供する *supplier* との間で，*client* の実行中のルーチンにおいて不変条件が成立していない状態から *supplier* が呼び出され，その *supplier* が *client* を呼び返す場合を，*Dependent Delegate Dilemma* という名の問題として詳しく説明している [65]．不変条件が成立していない状況で再入が起きた場合には，不変条件を検査することは無意味だが，従来の不変条件の検査方法では，これを検査し，不変条件違反の誤警告を生じる，という本章に関連する問題である．対処方法としては，*supplier* から呼び返される *client* のルーチンである *dependent delegate callback* については不変条件を検査しないことを提案している．

JML[60] は，メソッドの仕様に不変条件を追加しないことを示す helper プラグマを提供する．JML をサポートする拡張静的チェッカは，helper プラグマが付与されたメソッドについては，不変条件を検査しない．helper プラグマのような DbC 記法は本研究で用いた VARVEL ツールにおいても必要となる．一方，これらの手法・記法は，不変条件が成立しない状況で再入が起こる場合のみを考慮しており，再入時にも不変条件が成立すべき場合については考慮していない．本章の DbC 記法と検査方法は，再入時に不変条件が成立することを検査できる．

Barnett らは，C#を拡張した Spec#に関して，あるクラスのフィールドが別のクラスの不変条件に影響する場合に，別のクラスから見て当該フィールドを更新してよければ，自身の不変条件が成立し得なくてよいことを宣言し，それから当該フィールドに値を代入し，不変条件が成立するようにしてから，不変条件が成立した状態になったことを宣言する，というフィールドの更新に関わるクラス間のプロトコルを提案している [10]．フィールド更新の前後で不変条件が成立し得ない状況か否かをプログラマが明示するので，再入時に不変条件違反を誤検出することがない．しかし，この手法では，クラス間の依存性と提案されたプロトコルを熟知して，不変条件の成立不成立を明示する必要があり，プログ

ラマへの負担が大きい．さらに，静的解析では検証できないことがあり，代わりに実行時検査を必要とすることがある [12]．本研究では，プログラマはコーディング時点では C 言語の式で表現できる不変条件とそのスコープを決めればよく，負担は少ない．また，コールシーケンスをモジュールとする指示により，有界モデル検査ツールによる静的な検証が行える．

Fähndrich らは，再入が発生する呼出しをポインタ解析によって特定し，再入時の不変条件違反を単純なデータフロー解析によって検出する検査方法を提案している [35]．しかし，保守的な (conservative) なポインタ解析とデータフロー解析を用いるため，見かけの不具合を検出することがある．本研究では，再入箇所を含むコールシーケンス全体を対象として再入が起きる場合でも不変条件の検査を行う．さらに精度の高い有界モデル検査を用いるので，見かけの不具合を検出することは少ない．

第 5 章

関数ポインタに関する検査

5.1 背景

産業界で普及している C プログラムでは，関数ポインタを介した間接的な関数呼び出しが広く用いられている．間接呼び出しとは，プログラムの実行位置を示すプログラムカウンタに関数ポインタ変数の値を設定することにより，その値を先頭アドレスとする関数を実行する仕組みである．デバイスドライバや GUI フレームワークといったコールバック型プログラムでは，関数ポインタを介した間接呼出しを多用する．

関数ポインタを介して呼ばれる関数は，アドレスを関数ポインタに代入することにより実行時に決まり，その DbC 仕様を静的に特定することは難しい．また，呼出し元と呼ばれる側の関数が分業して開発される場合，一方の関数の開発者は，他方の関数の宣言と DbC 仕様しか参照できないことがある．

第 3 章では，有界モデル検査を用いたソフトウェアモデル検査にモジュラー検証の考え方を採用した．VARVEL ツールとして実現されている．しかし，DbC の基本機能 (3.2 節) によるモジュラー検証だけでは，関数ポインタを用いたプログラムを検証することは難しい．モジュラー検証を行う上で，関数ポインタに因る誤警告を減らす工夫が必要である．本章では，関数ポインタを用いるプログラムを，振舞いサブタイピング [61] の考え方に則って検査する方法を提案する [46] .

例えば，図 5.1 で，ファイル `A` の関数 `callTask` は，その引数 `n` が `0` 以上の場合に，`n` を引数として大域変数である関数ポインタ `task` を介した関数呼出しを行う (行 12,13).

ファイル A

```
10 : void (*task)( int );
11 : void callTask( int n ){
12 :   if ( 0 <= n )
13 :     (*task)( n );
14 : }
```

ファイル B

```
20 : void execTask(){
21 :   task = doTask;
22 :   callTask( 0 );
23 : }
24 : /** @pre  0<x   */
25 : void doTask( int x ){
26 :   int r = 100 / x ;
27 : }
```

図 5.1　関数ポインタを用いるプログラムの例

ファイル B の関数 execTask は，task に関数 doTask のアドレスを設定し，0 を引数として callTask を呼び出す (行 21,22)．ファイル A と B を与えると，VARVEL はポインタ解析により task を介して呼ばれる doTask を特定し，execTask のコードに callTask と doTask のコードをインライン展開する．VARVEL は doTask の事前条件 (0<x, 行 24) をプリミティブ (__assert(0<x)) に変換して，doTask の開始箇所に挿入し，有界モデル検査により事前条件への反例を出力する．しかし，ファイル B だけを与えると，VARVEL は execTask と doTask を独立に検査することになり，前述の事前条件への反例を出力できない．

この問題に対して，本章では，関数ポインタ自体に機能仕様（仮 DbC 仕様）を定義する．そして，検査対象のプログラムに関して次の二つの条件を考える．

(1) 関数ポインタを介した関数呼出し箇所において，仮 DbC 仕様が満たされるとの仮定の下で，検査対象のプログラムはプロパティ P を満たす．

(2) 関数ポインタを介して実際に呼ばれる可能性のあるすべての関数の機能仕様 (実 DbC 仕様) と仮 DbC 仕様とは置換可能性を持つ (実 DbC 仕様が成り立てば仮 DbC 仕様も成り立つ)．

この二つの条件がともに成り立てば，「関数群が実 DbC 仕様を満たすならば，検証対象のプログラムはプロパティ P を満たす」が成り立つ．この考え方に従って，仮 DbC 仕様の DbC 記法と，仮 DbC 仕様を用いたモジュラー検証と，実 DbC 仕様と仮 DbC 仕様の置換可能性検査からなる関数ポインタの検査方法とを提案する．この 2 段階の検査方法に

は，実際に呼ばれる関数本体のインライン展開が不要になるという長所がある．呼出し元の関数については (1) を，呼ばれる側の関数については (2) を検査すればよいので，これらの関数を分業して開発する場合にも検査を行いやすい．

関数ポインタに起因する別の問題の例を図 5.2 に示す．第 3.3 節の実験において，動的にメモリにロードされる関数に関して VARVEL が誤警告を発することを述べた．図 5.2 は，そのようなプログラムの例である．

```
int execdlfunc( int n ) {
  int (*comp)( int ) ;
  int result = 0 ;
  void *h = dlopen( "x.so", RTLD_LAZY ) ;
  comp = ( int (*)( int ) ) dlsym( h, "f" ) ;
  result = (*comp)( n ) ;
  dlclose( h ) ;
  assert( 0<=result );
  return result ;
}
```

図 5.2 動的にロードした関数を呼び出すプログラムの例

このプログラムを，標準的な有界モデル検査ツール CBMC(Version 4.0) で検査すると，関数ポインタを呼び出す行を削除してからモデル検査を行うため，不具合は検出されない．

```
> cbmc file.c --function execdlfunc
file file.c: Parsing
Converting
Type-checking file
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
**** WARNING: no body for function c::dlopen
**** WARNING: no body for function c::dlsym
**** WARNING: no body for function c::dlclose
size of program expression: 22 assignments
```

```
simple slicing removed 0 assignments
Generated 1 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
```

一方，VARVEL(Version 3.1.7) で検査すると，関数ポインタ comp を介した間接呼び出しを，未解釈関数 (pi_0) の呼び出しと見なし，非決定的に選ばれた任意の値を変数 result に代入する．そして，アサーション (0<=result) への違反を検出する．

```
> c_verify -t C -c -o . file.c -m execdlfunc
(HTML 出力の抜粋)
Step  Line  Function    Detailed Info
41    L50   execdlfunc  result {-1} = pi_0 {-1}
57    L52   execdlfunc  成立した条件 : 0 > result {-1}
58          execdlfunc  Witness shows: Assertion Violation (0 <= result)
```

このアサーション違反は，動的にロードされる関数の戻り値が 0 以上である場合は，誤警告である．第 3 章の実験では，このような誤警告が起きた．動的にロードされる関数のソースコードを検査に利用できないこと，関数ポインタ関して DbC 仕様を記述できないことが，誤警告の理由であった．

5.2 検査方法

関数ポインタを介した間接呼出しを行うプログラムでは，呼び出し元と呼ばれる側の関数が全く異なる時期に開発されることがある．本節では，関数ポインタに DbC 仕様 (仮仕様) を付与し，呼ばれる側の DbC 仕様 (実仕様) との置換可能性を定める．そして，次の 2 段階の検査によって，関数ポインタに関わるプログラムの検査を行う．

- **呼び出し元**
  - 仮仕様を用いたモジュラー検証
- **呼ばれる側**
  - 実仕様と仮仕様の置換可能性検査
  - 実仕様を用いたモジュラー検証

これによって，呼び出し元と呼ばれる側の関数がそれぞれ別の時期に開発される場合でも，それぞれを独立に検査し，合わせて，関数ポインタを介した呼出し全体の品質を保証できる．

5.2.1　仮仕様を用いたモジュラー検証

表 5.1　仮 DbC 仕様の記法

仕様の種別	説明
@variable_contract (f)	以降の事前・事後条件は関数ポインタ f に関する
@array_contract (f)	以降の事前・事後条件は関数ポインタ配列 f の各要素に関する
@array_elem_contract (f, i)	以降の事前・事後条件は関数ポインタ配列 f の i 番目の要素に関する
@type_contract (T)	以降の事前・事後条件は T 型の関数ポインタに関する
@field_contract (T, f)	以降の事前・事後条件は T 型構造体の関数ポインタフィールド f に関する
@field_array_contract (T, f)	以降の事前・事後条件は T 型構造体の関数ポインタ配列フィールド f の各要素に関する
@field_array_elem_contract (T, f, i)	以降の事前・事後条件は T 型構造体の関数ポインタ配列フィールド f の i 番目の要素に関する

組込み変数	説明
__aN (N=1..)	関数ポインタの N 番目の引数

仮 DbC 仕様の記法を表 5.1 に示す．仮 DbC 仕様は，まず関数ポインタ型の変数を指定し，その後に事前・事後条件を列挙する．図 5.3 の例では，関数ポインタ `task` について，事前条件を定義している (行 10a,10b)．産業界の C プログラムでは保守の観点から複雑な表現を用いないことが多いことから，表 5.1 の記法は，変数，1 次元配列，入れ子になっていない構造体といった単純な表現のみを対象としている．関数ポインタへのポインタや関数ポインタを引数に持つ関数ポインタなど，より複雑な表現には対応していない．5.3 節の実験では，本記法で問題なく DbC 仕様を記述できた．

VARVEL は，関数ポインタを介した関数呼出しを含むプログラムを，図 5.5 のアルゴリズム $check1$ によって検証する．$check1$ には，対象プログラム $prog$ と，$prog$ によって

ファイル A (仮 DbC 仕様の記述後)

```
10a: /** @variable_contract( task )
10b:       @pre 0<=__a1   */
10 : void (*task)( int );
11 : void callTask( int n ){
12 :   if ( 0<=n )
13 :     (*task)( n );
14 : }
```

図 5.3　仮 DbC 仕様の記述例

ファイル A (プリミティブへの変換後)

```
10 : void (*task)( int );
11 : void callTask( int n ){
12 :   if ( 0<=n )
13a:     __assert( 0<=n );
13b:     __assume( 0<=n );
13 :     (*task)( n );
14 : }
```

図 5.4　プリミティブへの変換例

間接的な関数呼出しに用いられる関数ポインタの仮 DbC 仕様を含むような関連プログラム $SRCs$ を与える．$check1$ は，まず，$SRCs$ から仮 DbC 仕様の集まり FCs を抽出し (行 1)，各仮 DbC 仕様ごとに，仮 DbC 仕様で指定された関数ポインタを介した関数呼出しの位置 loc を特定する (行 2,3)．次に，loc ごとに，表 5.2 の変換ルールに従って，仮 DbC 仕様をプリミティブを用いたプログラムコードに変換し，$prog$ に挿入する (行 4)．表 5.2 のルール a と b については，仮 DbC 仕様から変換されたプリミティブを loc の前に挿入し，ルール c と d については，loc の後に挿入する．最後に，$check1$ は，プリミティブが挿入されたプログラムに対してモデル検査を行い，その結果である $Result$ を返す (行 7)．$Result$ は，プロパティとそのプログラム上の位置，不具合の有無を示す情報，不具合がある場合は反例を含む．

例えば，図 5.3 で，対象プログラムとして関数 `callTask` を，関連プログラムとして

```
check1( prog, SRCs )
1.   FCs ← findFormalContracts( SRCs );
2.   foreach fc ∈ FCs do
3.     while ( loc ← findCall( fc, prog ) ) do
4.       prog ← insertPrimitives( loc, fc )
       endloop
     endloop
5.   return doSWMC( prog )
```

図 5.5 関数ポインタを介した関数呼出しを行うプログラムを検査するアルゴリズム

表 5.2 仮 DbC 仕様のソースコードへの変換ルール

	仕様の形式	ソースコード
a	`@pre P`	`__assert( P'); __assume( P');`
a'		`__assert( !n==i || P'); __assume( !n==i || P');`
b	`__old( v )`	`__old_v=v;`
c	`@param[out] v`	`v=__NONDET__();`
d	`@post Q`	`__assume( Q');`
d'		`__assume( !n==i || Q');`

ルール a' と d' は，仮 DbC 仕様が配列要素に関する場合に用いる
P': P に現れる仮引数を実引数で置換した式
n: プログラムコードにおける配列要素の添え字
i: 仮 DbC 仕様における配列要素の添え字
`__NONDET__()`: 未解釈関数．非決定的に選ばれた任意の値を返す
Q': Q に現れる仮引数を実引数で，式`__old( v )` を変数`__old_v` で置換した式

ファイル `A` を与えると，*check* は，**ファイル** `A` から関数ポインタ `task` についての仮 DbC 仕様 (行 10a,10b) を抽出し，`task` を介した関数呼出し `(*task)(n)` の位置 (行 13) を特定する．そして，*check*1 は変換ルール a に従い，仮 DbC 仕様の事前条件 `0<=__a1` の仮引数`__a1` を実引数 `n` に置き換えて，プリミティブを用いたプログラムコードに変換し，`task` を介した関数呼出しの前に挿入する (図 5.4 行 13a,13b)．プリミティブ挿入後の `callTask` をモデル検査により調べると，*assert* プリミティブ (図 5.4 行 13a) への反例は出力されない，すなわち，対象プログラムは，仮 DbC 仕様を守って，関数ポインタを介した関数呼出しを行っている．

5.2.2 実仕様と仮仕様の置換可能性検査

振舞いサブタイピングでは，オブジェクト指向言語において，スーパータイプの事前条件がサブタイプの事前条件を満たし，かつ，サブタイプの事後条件がスーパータイプの事後条件を満たす場合に，スーパータイプをサブタイプに置換できる [61]．本稿では，手続き型言語 C における関数ポインタと関数ポインタを介して呼ばれる関数の置換可能性を，振舞いサブタイピングにおけるスーパータイプとサブタイプの置換可能性と考える．すなわち，仮 DbC 仕様の事前条件は実 DbC 仕様の事前条件を満たし，実 DbC 仕様の事後条件は仮 DbC 仕様の事後条件を満たさねばならない．

$$Pre_{formal} \Rightarrow Pre_{actual} \qquad (3)$$

$$Post_{actual} \Rightarrow Post_{formal} \qquad (4)$$

```
check2( prog, SRCs )
1.  Result ← φ
2.  FCs ← findFormalContracts( SRCs )
3.  foreach fc ∈ FCs do
4.    while ( loc ← findNextAssignment( fc, prog ) ) do
5.      ap ← findAssignedProg( loc, prog )
6.      ac ← findActualContract( ap, SRCs )
7.      Result ← Result ∪ checkConsistency( ac, fc, loc )
      endloop
    endloop
8.  return Result
```

図 5.6　実・仮 DbC 仕様の置換可能性を検査するアルゴリズム

実 DbC 仕様と仮 DbC 仕様の置換可能性 (3)(4) は，図 5.6 に示すアルゴリズム *check*2 によって検査する．*check*2 には，対象プログラムとして関数ポインタを介した関数呼出しを行うプログラム *prog* と，*prog* によって関数呼出しに用いられる関数ポインタの仮 DbC 仕様と呼ばれる関数の実 DbC 仕様を含むような関連プログラム *SRCs* を与える．まず，*check*2 は，検査結果を格納する *Result* を空にする (行 1)．次に，*check*2 は，*SRCs* から仮 DbC 仕様の集まり *FCs* を抽出する (行 2-3)．*FCs* 中の各仮 DbC 仕様 *fc* ごとに，関数ポインタに関数のアドレスを設定する位置 *loc* を探し，*loc* ごとに関数ポインタにアドレスが設定される関数 *ap* の実 DbC 仕様 *ac* を探す (行 4-6)．そして，実仕様

ac と仮仕様 fc が置換しているというプロパティを制約解消問題として検査し，結果を $Result$ に追加する (行 7)．置換可能性の検査では，式 (3)(4) を否定した式をそれぞれ制約ソルバに与え，これらの式を満たすような変数への値の割当があるかどうかを調べる．割当があれば，それは置換可能性への反例である．$check2$ は一連の繰り返しを終えると，$Result$ を返す (行 10)．$Result$ は，プロパティ，関数ポインタを介した関数呼出しの位置，不具合の有無を示す情報，不具合がある場合は反例を含む．

例えば，対象プログラムとして図 5.3 の関数 `execTask` を，関連プログラムとして図 5.3 のファイル `A` と図 5.1 のファイル `B` を与えると，$check2$ は，ファイル `A` から関数ポインタ `task` の仮 DbC 仕様 (事前条件 `0<=__a1`，図 5.3 行 10b) を抽出する．$check2$ は，`execTask` のコードから `task` に関数アドレスが設定される位置 (図 5.1 行 21) を見つけ，アドレスが設定される関数 `doTask` の実 DbC 仕様 (事前条件 `0<x`，図 5.1 行 24) を見つける．$check2$ が見つけた事前条件に関する実 DbC 仕様 `0<x` と仮 DbC 仕様 `0<=__a1` の置換可能性を検査するためのスクリプト (SMT-LIB 形式) を図 5.7 に示す．このスクリプトは，関数 `doTask` の引数 `x` と関数ポインタ `task` の引数`__a1` を同じ名前の引数 `a1` とした実 DbC 仕様 (`0<a1`) と仮 DbC 仕様 (`0<=a1`) との間で，式 (3) が成立しないような引数の存在を調べる内容となっている．図 5.7 のスクリプトを SMT ソルバ Yices [34] に与えて，論理演算と線形整数演算の範囲で制約を解くと，変数値の割当 `(= a1 0)` を得る，すなわち，第 1 引数が 0 の場合に，仮および実 DbC 仕様の事前条件の間に置換可能性がないことが判る．

```
                                ; (; から行末まではコメント)
1 :logic QF_LIA                 ; 限量子なしの線形整数演算
2 :extrafuns (( a1 Int ))       ; 第 1 引数
3 :assumption                   ; 置換可能性を破る条件
4 (not (implies                 ;   含意の否定
5   (<= 0 a1)                   ;   含意の前件 : 仮事前条件 0<=__a1
6   (<  0 a1)))                 ;   含意の後件 : 実事前条件 0<x
```

図 5.7　実・仮 DbC 仕様の置換可能性検査スクリプトの例

5.3 実験

5.2.1 節と 5.2.2 節で述べた，関数ポインタを持つプログラムを検証する提案方法を，教育用 OS である MINIX(version 3.1.1) に適用した．MINIX はマイクロカーネルのアーキテクチャを採用しており，プロセス間メッセージ通信を多用する．信頼性を要求されるシステムソフトウェアであること，ソースコードが公開されていること，文献 [79] に従って DbC 仕様を記述できることから，モジュラー検証の実験対象として適切と考えた．OS 本体のプロセスは，サーバ，デバイスドライバ，カーネルの 3 層に分けられる．本実験では，表 5.3 に示す各層の機能から，関数ポインタを介した関数呼出しを行う関数を含むファイルを選んだ．表 5.3 の列において，サイズ，関数，関数ポインタの形態は，ファイルの行数，関数の個数，関数ポインタが C 言語のどの言語要素として宣言されたかを示す．なお，MINIX のヘッダファイル中の関数ポインタを含む宣言 75 個を調べたところ，これらの宣言は表 5.1 の記法で仕様を記述し得るものであった．また，表 5.3 で選んだファイルに含まれる関数は全て，表 3.1 と 5.1 の記法により仕様を記述できた．

表 5.3　実験対象としたソースコード

層	機能	サイズ	関数	関数ポインタの形態
サーバ	ファイルシステム	718	29	構造体フィールド
ドライバ	フロッピーディスク	763	37	構造体フィールド
カーネル	システムコール	1107	38	配列

本実験では，まず，DbC 仕様として事前・事後条件のみを記述し，モジュラー検証を行って，誤警告の発生を調べた．次に，不変条件と関数ポインタの仮 DbC 仕様を追加し，モジュラー検証と実・仮 DbC 仕様の置換可能性検査を行った．モジュラー検証にはソフトウェアモデル検査ツール VARVEL(Version 3.1.7) を用いた．1 関数あたりの検査の上限時間と BMC の最大メモリ使用量として，それぞれ 10 分と 1G バイトを指定した．DbC 仕様の置換可能性検査には SMT ソルバ Yices(Version 2) を論理演算と線形整数演算の範囲で用いた．いずれの検査も，CPU クロックは 3.4GHz，メモリ容量は 2G バイトの Linux 機上で実行した．

5.3.1 事前・事後条件のみを用いたモジュラー検証

実験対象の各関数には，文献 [79] とソースコードを参考に，引数と戻り値について DbC 仕様を記述した．VARVEL を用いたモジュラー検証の結果を表 5.4 に示す．表 5.4 の列において，Pre，Post，Alarm はそれぞれ事前条件，事後条件，警告の個数である．続く，Bug，GV，FP，Others は警告の内訳であり，それぞれ不具合と考えられる警告，大域変数が非決定的に選ばれた任意の値を取り得るという近似による誤警告，関数ポインタを介した関数呼出しに副作用は無く，かつ戻り値は非決定的に選ばれた任意の値を取り得るという近似による誤警告，その他の近似による誤警告の個数である．共通ヘッダ層は他層から参照されるヘッダファイルの集まりであり，関数宣言と DbC 仕様を含む．

表 5.4 事前・事後条件のみを用いたモジュラー検証の結果

層	Pre	Post	Alarm	Bug	GV	FP	Others
サーバ	46	37	20	2	11	6	1
ドライバ	41	18	2	1	0	1	0
カーネル	77	39	3	2	0	1	0
共通ヘッダ	37	71					

大域変数に因る誤警告はサーバ層で多く発生した．例えば，MINIX においてプロセス番号は-4 以上であるが，VARVEL はプロセス番号を表す大域変数が-4 未満を取り得るとし，プロセス番号を引数とする関数呼出しで事前条件違反を警告した．また，関数ポインタを介した関数呼出しは，各層において誤警告の原因となった．例えば，応答メッセージを要求メッセージと同じ構造体変数で受け取る場合には構造体フィールドの値が変更されることがあるが，関数ポインタを介した関数呼出しには副作用が無い，すなわち構造体フィールドの値は変らないとするために，後続のソースコードを正しく解析できなかった．この問題の解決方法を次節に提案する．

5.3.2 不変条件・仮仕様を用いたモジュラー検証

節 5.3.1 の誤警告を削減するために，大域変数の値を制限する不変条件と，関数ポインタの仮 DbC 仕様を追記した．関数ポインタに設定された関数アドレスを調べ，これらの

関数の DbC 仕様と仮 DbC 仕様が式 (3)(4) を満たすように，仮 DbC 仕様を決めた．また，関数ポインタを介した関数呼出しの位置には，仮 DbC 仕様に対応したプリミティブを挿入した．

表 5.5　不変条件と仮 DbC 仕様を追加したモジュラー検証の結果

層	Inv	Spec	Prim	Alarm	Bug	Others
サーバ	11	0	69	11	6	5
ドライバ	0	29	38	1	1	0
カーネル	0	22	22	20	20	0
共通ヘッダ	0	17				

VARVEL を用いたモジュラー検証の結果を表 5.5 に示す．表 5.5 の列において，Inv，Spec，Prim，Alarm はそれぞれ不変条件，仮 DbC 仕様，仮 DbC 仕様から変換されたプリミティブ，警告の個数である．続く，Bug，Others は警告の内訳であり，それぞれ，不具合と考えられる警告，大域変数および関数ポインタ以外の近似による誤警告の個数である．

不変条件と仮 DbC 仕様に対応したプリミティブを追加したことにより，警告は増えたが，誤警告は減っている．警告の正しさを「不具合数/警告数」とすると，表 5.4 での 20%(5/25) に対して表 5.5 では 84%(27/32) と改善している．大域変数によるデータの受け渡しを行うサーバ層においては不変条件によって，関数ポインタを介した関数呼出しを行う各層においては仮 DbC 仕様によって，誤警告を削減できたといえる．

```
/**
 @pre  op==CANCEL ||
       op==DEV_READ || op==DEV_WRITE || op==DEV_IOCTL ||
       op==DEV_SCATTER || op==DEV_GATHER
 @pre  ((dev>>MAJOR)&BYTE)<NR_DEVICES
 @pre  -NR_TASKS<=proc && proc<NR_PROCS
 @post !( op == CANCEL ) || ( __return <= OK )
*/
int dev_io( int op, dev_t dev, int proc,
            void *buf, off_t pos, int bytes, int flags)
{
  struct dmap *dp;
```

```
message dev_mess;

dp = &dmap[(dev >> MAJOR) & BYTE];
if (dp->dmap_driver == NONE) return ENXIO;

dev_mess.m_type   = op;
dev_mess.DEVICE   = (dev >> MINOR) & BYTE;
dev_mess.POSITION = pos;
dev_mess.PROC_NR  = proc;
dev_mess.ADDRESS  = buf;
dev_mess.COUNT    = bytes;
dev_mess.TTY_FLAGS = flags;

/* 間接呼び出しの事前条件に相当するプリミティブ */
__assert( -NR_TASKS<=dp->dmap_driver && dp->dmap_driver<NR_PROCS );
__assume( -NR_TASKS<=dp->dmap_driver && dp->dmap_driver<NR_PROCS );
__assert( &dev_mess != NULL );
__assume( &dev_mess != NULL );
__assert( CANCEL <= (&dev_mess)->m_type && (&dev_mess)->m_type <= DEV_STATUS );
__assume( CANCEL <= (&dev_mess)->m_type && (&dev_mess)->m_type <= DEV_STATUS );
__assert( !( (&dev_mess)->m_type != DEV_STATUS ) ||
  ( 0 <= (&dev_mess)->DEVICE && (&dev_mess)->DEVICE <= BYTE ) );
__assume( !( (&dev_mess)->m_type != DEV_STATUS ) ||
  ( 0 <= (&dev_mess)->DEVICE && (&dev_mess)->DEVICE <= BYTE ) );
__assert( !( (&dev_mess)->m_type != DEV_STATUS ) ||
  ( -NR_TASKS<=(&dev_mess)->PROC_NR && (&dev_mess)->PROC_NR<NR_PROCS ) );
__assume( !( (&dev_mess)->m_type != DEV_STATUS ) ||
  ( -NR_TASKS<=(&dev_mess)->PROC_NR && (&dev_mess)->PROC_NR<NR_PROCS ) );
/* 事後条件で参照される変数について，開始時点の値の退避 */
__oi1 = (&dev_mess)->m_type;
__oi2 = (&dev_mess)->m2_i1;
__oi3 = (&dev_mess)->m2_i2;

/* 1 箇所目の間接呼び出し */
(*dp->dmap_io)(dp->dmap_driver, &dev_mess);
(&dev_mess)->m_type=__NONDET__();
(&dev_mess)->m2_i1=__NONDET__();
(&dev_mess)->m2_i2=__NONDET__();

/* 間接呼び出しの事後条件に相当するプリミティブ */
/*    メッセージ通信に成功した場合 */
__assume( !( (&dev_mess)->m_type != __oi1 ) ||
```

```
    ( DEV_REPLY <= (&dev_mess)->m_type && (&dev_mess)->m_type <= DEV_NO_STATUS ||
      (&dev_mess)->m_type == TASK_REPLY ) );
  __assume( !( (&dev_mess)->m_type != __oi1 && __oi1 != DEV_STATUS ) ||
    ( (&dev_mess)->REP_PROC_NR == __oi3 ) );
  __assume( !( (&dev_mess)->m_type != __oi1 && __oi1 == CANCEL ) ||
    ( (&dev_mess)->REP_STATUS <= OK ) );
  __assume( !( (&dev_mess)->m_type != __oi1 && __oi1 == DEV_OPEN ) ||
    ( (&dev_mess)->REP_STATUS <= BYTE ) );
  __assume( !( (&dev_mess)->m_type != __oi1 && __oi1 == DEV_CLOSE ) ||
    ( (&dev_mess)->REP_STATUS <= OK ) );
  /*   メッセージ通信に失敗した場合 */
  __assume( !( (&dev_mess)->m_type == __oi1 ) ||
    ( (&dev_mess)->REP_PROC_NR == __oi2 ) );
  __assume( !( (&dev_mess)->m_type == __oi1 ) ||
    ( (&dev_mess)->REP_STATUS == __oi3 ) );

  if (dev_mess.REP_STATUS == SUSPEND) {
    if (flags & O_NONBLOCK) {
      dev_mess.m_type = CANCEL;
      dev_mess.PROC_NR = proc;
      dev_mess.DEVICE = (dev >> MINOR) & BYTE;
        :
      /* 1 箇所目の間接呼び出しと同様の事前条件に相当するプリミティブ */
        :
      /* 2 箇所目の間接呼び出し */
      (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
        :
      /* 1 箇所目の間接呼び出しと同様の事後条件に相当するプリミティブ */
        :
      if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
    } else {
      suspend(dp->dmap_driver);
      return(SUSPEND);
    }
  }
  return(dev_mess.REP_STATUS);
}
```

図 5.8　不具合を含むソースコードの例

図 5.8 は，仮 DbC 仕様を用いたモジュラー検証で見つかった不具合を含むソースコー

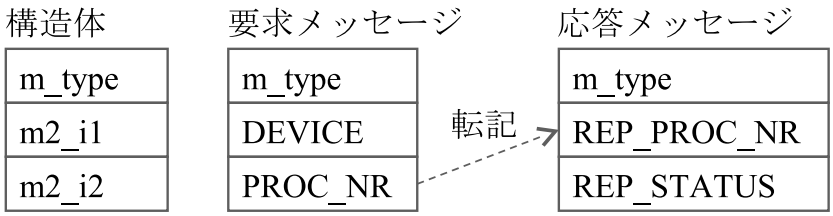


図 5.9 MINIX のデバイスドライバの要求応答メッセージ

ドの一例である．関数ポインタ dp->dmap_io を介した関数呼出しは，要求メッセージの送信と応答メッセージの受信を行う．MINIX のメッセージ通信の様子を図 5.9 に示す．要求と応答メッセージは同じ構造体変数に格納される．構造体を要求メッセージとして利用する場合には，フィールド m2_i1 と m2_i2 にそれぞれ別名 DEVICE と PROC_NR でアクセスする．DEVICE には処理の依頼先であるデバイスの副番号 (0 以上 BYTE(255) 以下) が，PROC_NR には要求元のプロセス番号 (-NR_TASKS(4) 以上 NR_PROCS(100) 未満) が設定される．応答メッセージとしての利用では，フィールド m2_i1 と m2_i2 にそれぞれ別名 REP_PROC_NR と REP_STATUS でアクセスする．REP_PROC_NR には要求元のプロセス番号が転記され，REP_STATUS にはデバイスによる処理結果が設定される．メッセージの送受信に失敗した場合には，構造体の内容は変らない．すなわち，dp->dmap_io を介した関数呼出しでは，要求メッセージが設定された構造体 dev_mess を用いてメッセージ通信を行い，通信に成功した場合は dev_mess の内容は応答メッセージとなり，通信に失敗した場合は要求メッセージのままとなる．関数ポインタ dp->dmap_io の仮 DbC 仕様 (事後条件) に対応したプリミティブは，通信成功の場合は dev_mess の内容が図 5.9 に示すように変り，失敗の場合は dev_mess の内容が変らないことを表す．

検出された不具合は，次のようなものである．関数 dev_io の引数 op は事前条件を満たす値 CANCEL であり，引数 proc は事前条件を満たす 0 より大きい値とする．proc の値は要求メッセージのフィールド m2_i2(別名 PROC_NR) に設定される．関数ポインタ dp->dmap_io を介した関数呼出しで，通信に失敗した場合，m2_i2 の値は変らず 0 より大きい値である．dev_io が m2_i2(別名 REP_STATUS) を戻り値として返すと，引数 op が CANCEL であれば，戻り値は OK(値は 0) 以下であるという事後条件に違反する．

5.3.3 仕様の置換可能性検査

関数ポインタの値を初期化する変数宣言と関数ポインタに値を代入する文ごとに，節 5.2.2 に示した実 DbC 仕様と仮 DbC 仕様との置換可能性を検査するための制約ソルバ用のスクリプトを記述する．本実験では，制約ソルバとして Yices [34] を用いて検査を行った．検査結果を表 5.6 に示す．

表 5.6　実 DbC 仕様と仮 DbC 仕様の置換可能性検査の結果

層	代入箇所	Pre 反例	Post 反例
サーバ	7	0	0
ドライバ	12	0	0
カーネル	29	0	0

表 5.6 の列において，代入箇所，Pre 反例，Post 反例はそれぞれ関数ポインタに実際の関数のアドレスを代入する箇所の個数，式 (3) が成立しない場合の警告の個数，式 (4) が成立しない場合の警告の個数である．設定位置毎に実 DbC 仕様と仮 DbC 仕様の置換可能性を検査したところ，置換可能性への反例は見つからなかった．置換可能性 (3)(4) は DbC 仕様の設計にも注意を向ける効果があったと言える．

5.4 考察

本章では，関数ポインタを介した間接呼び出しを通常の関数呼び出し同様に検証するために，関数ポインタの DbC 記法を導入し，置換可能性検査およびモジュラー検査の 2 段階の検査アプローチを考案した．また，MINIX を対象とした実験を行い，仮仕様の導入による誤警告の軽減と，間接呼び出しに関連した不具合の発見という提案方法の効果を実証した．

産業界の C プログラムには，GUI フレームワークやデバイスドライバなどのコールバック型プログラムが多く見受けられ，関数ポインタが多用されている．本研究によって，C プログラムのモジュラー検証は実用化に近付いたと考える．

既存の C プログラムの DbC 記法である ACSL [13] には，関数ポインタに事前・事後

条件を定義する記法がない．ACSL にも，関数ポインタに DbC 仕様を定義する記法が追加されることが望ましい．

5.5　関連研究

C プログラムにおける関数ポインタは，高階関数の実装に相当する．Scheme では，高階関数についての DbC 仕様の提案がある [37]．ただし，検査はテスト実行によって行う．本章では，Liskov の振る舞いサブタイピングの考え方に則って，関数ポインタの DbC 仕様 (仮仕様) と関数ポインタを介して呼ばれる関数の DbC 仕様 (実仕様) の置換可能性を定め，関数ポインタに関わるプログラム全体の検査を次の 2 段階の検査に分けるアプローチを提案した．

- **呼び出し元**.
  - 仮仕様を用いたモジュラー検証
- **呼ばれる側**.
  - 実仕様と仮仕様の置換可能性検査
  - 実仕様を用いたモジュラー検証

第 6 章

ソフトウェアモデル検査とテストケース生成の統合

6.1 背景

ソフトウェアの信頼性向上に関心が高まっている．従来からの信頼性向上手段であるプログラム・テスト技術は，プログラム実行に依るので，テストケース毎に１つの経路しか検査せず，網羅度が低い．ソフトウェアモデル検査はロジック・モデル検査のプログラム自動検証への適用である．ロジック・モデル検査は全ての経路を考慮した網羅探索を行う [23]．その網羅度の高さから，ソフトウェアモデル検査は，従来のテスト技術に代わるものとして有望である．なお，産業界では，信頼性の基準はテストで与えられており，自動検証ツールを使う場合であっても，テストによる検査との関係を論じる必要がある．

モデル検査には，状態爆発と呼ばれるスケーラビリティの問題があり，状態や状態遷移が大規模なプログラムについては，不具合の有無を判定できないことがある．状態数は，抽象化に基づく過大近似の導入により削減できる [22]．しかし，見かけの経路が増えることにより，実際には起こり得ない不具合を検出するという誤警告の問題を起こす．有界モデル検査法 (BMC) [18] は，探索範囲を一定の深さまでに限定し，不具合を効率よく検出する手法である．探索範囲を限定することにより，状態爆発を抑える．BMC をプログラム検証に適用する場合，対象プログラムを探索範囲内に制限する過小近似が必要とな

る．範囲外の不具合を見逃す問題があるので，過小近似の導入には注意が必要である．

本章では，有界モデル検査法を用いたモジュラー検証において，有界モデル検査を可能とするためのプログラムへの近似導入による問題を，テストによって補うツールを提案する [47] [48]．提案方法では，モデル検査による警告が誤っていないことを，警告についての反例から生成したテストケースを実行して判定する．また，不具合の見逃しの可能性を，モデル検査の方法を応用したトラップ挿入によって自動検知する．トラップはモデル検査とテストに共通のカバレッジ基準に則って挿入する．モデル検査をテスト実行で補うことにより，ソフトウェアモデル検査で導入される近似の影響がなくなり，網羅度の高い検査ができる．実際に，MINIX のソースコードを用いた実験により，モデル検査とテスト実行とを合せて，網羅度の高い検査が実施できた．

6.1.1 テスト技術とモジュラー検証

プログラム・テスト技術はソフトウェアの品質を検査する手段の 1 つである．産業界では，従来から広く実践されており，現在も品質確保の中心的な役割を担っている [78]．テスト技術は，プログラム実行に依るので，１つのテストケース毎に１つの経路しか検査せず，網羅度が低い．テストケースをどう選ぶか，テスト実行をどこまで行うかは，カバレッジ基準を定めて，これを満足するように行う．ブラックボックスのテスト技術においては，プログラムの内部構造は考えずに，入力データに着目して，テストケースを選ぶ．同値分割や境界値分析によって，同じ結果をもつと期待されるテストケースを考える．ホワイトボックスのテスト技術においては，プログラムの内部構造を考慮し，それぞれの文や分岐を実行するようにテストケースを選ぶ．

しかし，カバレッジ基準を満たす検査を，実際に行うことは難しい．図 6.1 のプログラム `twoMalloc` を用いて説明する．`twoMalloc` は，本来は引数 `p1` と `p2` のいずれかが `NULL` であるか，関数 `malloc` を用いた動的なメモリ割り当てに失敗した場合は `0` を返し，動的なメモリ割り当てに成功した場合は `1` を返す．しかし，図 6.1 では，`malloc` の 2 回目の呼出しに成功したかどうかを判断しない不具合がある．

ホワイトボックス・テストの観点として，例えば全テストケースを終えた時に各文を少なくとも 1 回は実行するという命令カバレッジ基準に則ると，図 6.2 のテストケースが

```
 4: /**
 5:  @post (p1==NULL && __return==0) || \
 6:        (p2==NULL && __return==0) || \
 7:        (p1!=NULL && *p1==NULL && __return==0 ) || \
 8:        (p2!=NULL && *p2==NULL && __return==0 ) || \
 9:        (p1!=NULL && *p1!=NULL && p2!=NULL && *p2!=NULL && __return==1 )
10: */
11: int twoMalloc( int **p1, int **p2 ){
12:   if ( p1==NULL ) return 0;
13:   if ( p2==NULL ) return 0;
14:   *p1=(int*)malloc( sizeof(int) );
15:   if ( *p1==NULL ) return 0;
16:   *p2=(int*)malloc( sizeof(int) );
17:  /* BUG : if ( *p2==NULL ) return 0; */
18:   return 1;
19: }
```

図 6.1　malloc を 2 回呼ぶプログラムの例

考えられる．不具合のあるプログラムを元にテストケースを作っているため，2 回目の malloc に失敗する場合を考慮できていない．プログラムの構造は網羅できるかもしれないが，本来の機能に対しては，不十分なテストケースといえる．

ブラックボックス・テストの観点として，例えば同値分割を行うと，図 6.3 のテストケースが考えられる．近年，コンピュータへのメモリの搭載量が増えており，関数 malloc は失敗することは少ない．ケース b3 と b4 を実施するには，テストケースごとに適切に失敗する (NULL を返す) ような malloc のスタブを用意する必要がある．対象プログラムが別の関数を呼び出し，その内部で malloc を呼び出している場合，スタブを提供できないことがある．実際には実行し難いことから，不十分なテスト結果となることがある．

図 6.2　ホワイトボックス・テストの観点のテストケース例

ケース	入力	出力	備考
w1	p1==NULL, p2!=NULL	戻り値==0	行 1 の return 文までを実行
w2	p1!=NULL, p2==NULL	戻り値==0	行 2 の return 文までを実行
w3	p1==NULL, p2!=NULL	*p1==NULL, 戻り値==0	行 4 の return 文までを実行
w4	p1!=NULL, p2!=NULL	*p1!=NULL, *p2!=NULL, 戻り値==1	最後まで実行

図 6.3　ブラックボックス・テストの観点のテストケース例

ケース	入力	出力	備考
b1	p1==NULL, p2!=NULL	戻り値==0	不正引数 p1
b2	p1!=NULL, p2==NULL	戻り値==0	不正引数 p2
b3	p1!=NULL, p2!=NULL	戻り値==0	1 回目の malloc に失敗
b4	p1!=NULL, p2!=NULL	戻り値==0	2 回目の malloc に失敗
b5	p1!=NULL, p2!=NULL	*p1!=NULL, *p2!=NULL, 戻り値==1	成功

モジュラー検証を行う場合には，図 6.3 に相当する DbC 仕様をプログラムに追記する．図 6.1 の行 5–9 は，それぞれ図 6.3 のケース b1–b5 に対応する．有界モデル検査を用いたモジュラー検証ツール VARVEL(第 3 章) によって，プログラム twoMalloc を検査すると，2 回目の malloc が失敗し NULL を返した場合に，twoMalloc 自体の戻り値は期待される 0 ではなく 1 であり，下記のような事後条件違反の警告が出力される．

```
(VARVEL のコマンド実行．-t C オプションはモジュラー検証を指示する．
変数 x の値が v であることを x{v}はで表す)
> c_verify -t C -c -o . utsample.c -m twoMalloc
(HTML 出力の抜粋)
Step  Line  Function   Detailed Info
 0    L11   twoMalloc  p1 =47
 0    L11   twoMalloc  p2 =63
13    L12   twoMalloc  成立した条件 : p1 {47} > 0
15    L13   twoMalloc  成立した条件 : p2 {63} > 0
26    L14   malloc     __return {93} = ヒープで確保した領域
33    L14   twoMalloc  *p1 {93} = ( 戻り値 {93} )
36    L15   twoMalloc  成立した条件 : *p1 {93} > 0
47    L16   malloc     __return {0} = 0
51    L16   twoMalloc  *p2 {0} = ( 戻り値 {0} )
53    L18   twoMalloc  戻り値 {1} = 1
59    L5    twoMalloc  成立した条件 : p1 {47} > 0
62    L5    twoMalloc  成立した条件 : p2 {63} > 0
65    L5    twoMalloc  成立した条件 : p1 {47} > 0
67    L5    twoMalloc  成立した条件 : *p1 {93} > 0
70    L5    twoMalloc  成立した条件 : p2 {63} > 0
72    L5    twoMalloc  成立した条件 : *p2 {0} <= 0
74    L5    twoMalloc  成立した条件 : __return {1} != 0
77    L5    twoMalloc  成立した条件 : p1 {47} > 0
79    L5    twoMalloc  成立した条件 : *p1 {93} > 0
81    L5    twoMalloc  成立した条件 : p2 {63} > 0
83    L5    twoMalloc  成立した条件 : *p2 {0} <= 0
84    L5    twoMalloc  __decision1 {0} = 0
86    L5    twoMalloc  成立した条件 : __decision1 {0} == 0
87          twoMalloc  Witness shows: Assertion Violation (... 事後条件の
式...)
```

ブラックボックス・テストの観点からは DbC 仕様の事前条件を満たす入力データを網羅

し，ホワイトボックス・テストの観点からはプログラムから導出した有限状態遷移系上の経路を網羅探索するので，1 回の検査で，従来のテスト作業以上の検査を実施するといえる．

さらに，メモリ操作関数のような外部環境の影響がない場合でも，テスト実行では不具合を発見できないことがある．図 6.4 のプログラム coverdButMissed において，引数 n に 0 と 4 を与えると，分岐 1<=n はそれぞれ偽と真になり，分岐カバレッジ基準を満たす．また，戻り値はそれぞれ 0 と 4 となり，事後条件を満たす．分岐カバレッジ基準を満たすというホワイトボックス・テストの観点と，事後条件の場合分けを尽くすというブラックボックス・テストの観点のいずれからも，問題は見つからない．

```
1: /**
2:  @post __return<=0 || 4<=__return
3: */
4: int coverdButMissed( int n ){
5:   int n1;
6:   if ( 1<=n ) { n1 = n<<2; printf("WhiteBox: True block\n"); }
7:   else        { n1 = n;    printf("WhiteBox: Else block\n"); }
8:   return n;  /* Should return n1. */
9: }
```

図 6.4　分岐カバレッジ基準を満たしても不具合のあるプログラムの例

しかし，プログラム coverdButMissed は本来は変数 n1 を戻り値として返すべきところ，誤って引数 n 自体を返しているため，引数 n に 1 以上 3 以下の値を代入すると，事後条件に違反する．有界モデル検査ツール VARVEL を用いてモジュラー検証を行うと，1 回の検査で，この不具合を発見できる．

```
(VARVEL のコマンド実行．-t C オプションはモジュラー検証を指示する)
> c_verify -t C -c -o . utsample.c -m coverdButMissed
(HTML 出力の抜粋)
Step  Line  Function         Detailed Info
 0    L4    coverdButMissed  n =3
24    L8    coverdButMissed  戻り値 {3} = n {3}
29    L2    coverdButMissed  成立した条件 : __return {3} > 0
```

```
31    L2    coverdButMissed  成立した条件 : 4 > __return {3}
32    L2    coverdButMissed  __decision1 {0} = 0
34    L2    coverdButMissed  成立した条件 : __decision1 {0} == 0
35          coverdButMissed  Witness shows: Assertion Violation
                               __return <= 0 || 4 <= __return
```

6.1.2 近似の問題

ソフトウェアモデル検査はテスト作業を自動化する有望な手段である．しかし，産業界では，信頼性の基準はプログラム・テスト技術で与えられており，たとえ自動検証ツールを使った場合であっても，テスト技術による検査との関係を論じる必要がある．

ソフトウェアモデル検査では，ロジック・モデル検査が適用できるように，対象プログラムを有限状態遷移システムに変換する．モデル検査において，状態爆発というスケーラビリティの問題が起きると，有限状態空間上の探索を終えられず，不具合の有無を示せない．変換の際に抽象化といった過大近似を導入することにより，状態や状態遷移の数を減らして状態爆発を緩和できる [22]．しかし，過大近似によって見かけの振舞いが増える．変換後のプログラムが元のプログラムの振舞いを全て含むような近似を過大近似と呼ぶ．逆に，過大近似されたプログラムは元のプログラムには存在しない見かけの経路を含む．プログラムが含む経路とその集合を π と Π，変換を行う関数を α とし，変換後のプログラムにおける経路・集合を $'$ で元のプログラムのそれと区別すると，過大近似は次のように表せる．

$\forall \pi \in \Pi \cdot \exists \pi' \in \Pi' \cdot \pi' = \alpha(\pi)$

$\exists \pi' \in \Pi' \cdot \pi' \notin ran(\alpha) \quad (ran(\alpha)$ は α の値域$)$

見かけの経路で検出される不具合は，元のプログラムでは起こり得ない見かけの不具合である．見かけの不具合はバグの原因を調査する必要がないので，誤警告であることを自動判定したい．モデル検査の反例から不具合を再現しえるテストケースを生成する研究 [20] [15] が判定に利用できる．

有界モデル検査法では，有限状態空間をある深さまでしか探索しないことにより，効率的に不具合を見つける．プログラムを探索範囲に制限するための工夫は，過小近似を導入することである．元のプログラムが変換後のプログラムの振舞いを全て含むような近似を

過小近似と呼ぶ．逆に，過小近似されたプログラムは元のプログラムの一部の経路を含まない．

$\forall \pi' \in \Pi' \cdot \exists \pi \in \Pi \cdot \pi' = \alpha(\pi)$

$\exists \pi \in \Pi \cdot \pi \notin dom(\alpha)$　($dom(\alpha)$ は α の定義域)

元のプログラムにしか存在しない経路 π 上のみで起こる不具合は，変換後のプログラムを探索しても検出できない．不具合の見逃しを防ぐためには，元のプログラムにおいてモデル検査では探索されない経路の有無を自動で検知したい．また，そのような経路が検知されれば，モデル検査とは別の手段で検査したい．別手段としてはプログラムテストを考えることができる．

テストを行うには，テストケース (入力データと期待される結果) を生成するために，対象プログラムの仕様が必要となる．事前・事後条件といった仕様を同値分割などにより区分けし，区分ごとにテストケースを生成する研究がある [32] [62] ．これを併用すればよい．

6.2 方式

ソフトウェアモデル検査による検査結果は，近似導入の影響を受けている可能性がある．本節では，ソフトウェアモデル検査とテストケース生成を組み合せて，近似導入の自動判定やカバレッジ基準を満たす検査を行う方法を提案する．一般に，過大近似と過小近似は混在しえるため，ある経路にどの近似が導入されたかを特定することは難しい．そこで，経路がモデル検査により探索可能か，テストにより実行可能か，という観点から，近似導入の有無を調べる．

過大近似については，モデル検査で得られた反例からテストケースを生成し，テストによって不具合の再現を試みる．再現しなければ，不具合は過大近似の影響による見かけのものであり，反例は誤警告であると自動判定できる．

過小近似については，モデル検査の探索経路に存在すれば必ず反例が得られるトラップを対象プログラムに埋め込む．反例が得られなければ，トラップを含む経路はある箇所から探索されていない，即ち，不具合の見逃しの可能性を自動検知できる．検知した場合に

は，DbC 仕様からテストケースを生成し，テストを行う．テストと共通のカバレッジ基準に則って，検知用のトラップを埋め込むことにより，プログラムの大部分をモデル検査により網羅探索し，残りの部分を少ないテストケースで補助的に検査するといった，従来のテストのみによる方法に比べて網羅度の高い検査ができる．

6.2.1　処理の概要

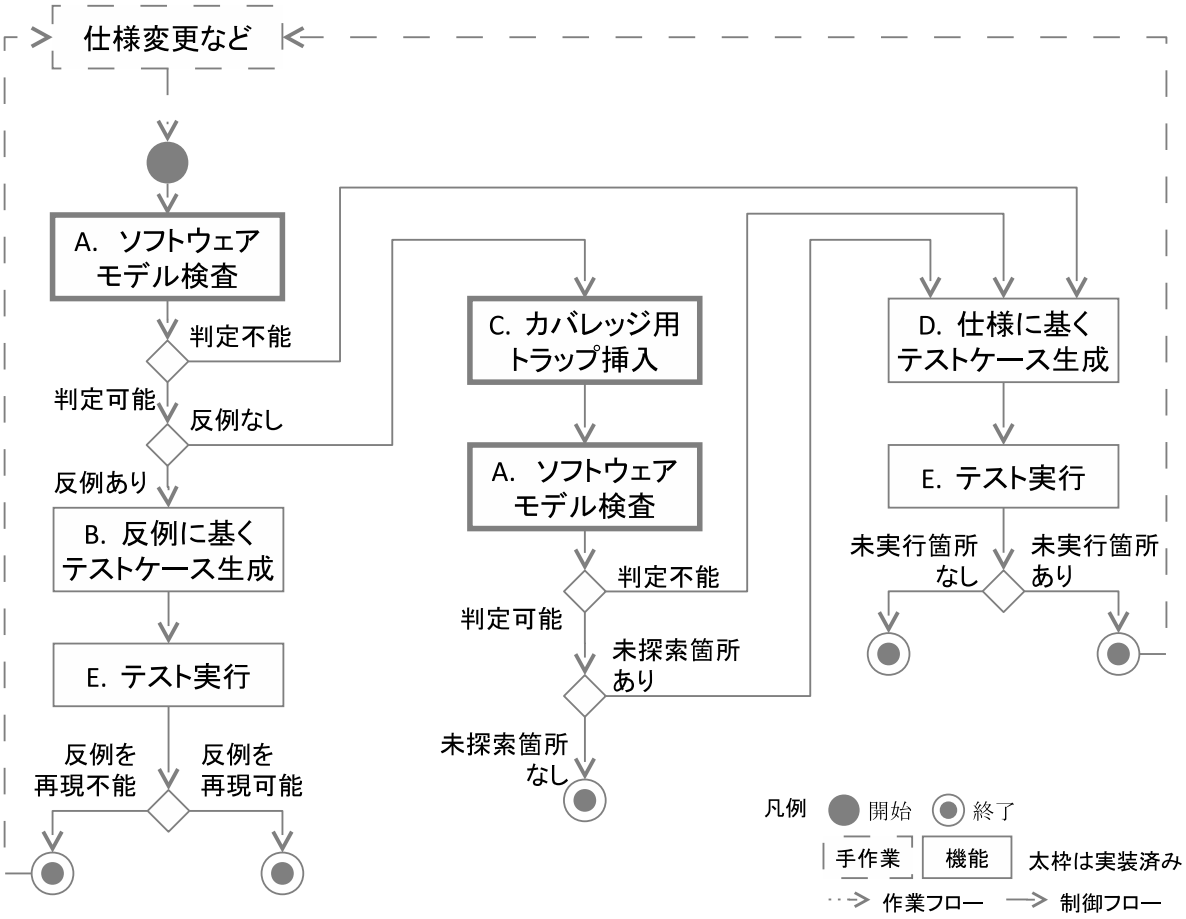


図 6.5　処理の流れ

本稿で提案する，有界モデル検査とテストケース生成を統合する方式における処理の流れを図 6.5 に示す．図 6.5 における開始ノードから各終了ノードまでは自動実行可能な処理である．この自動実行可能な一連の処理は，検査の対象プログラムについて有界モデル検査を行うソフトウェアモデル検査機能 A，有界モデル検査で得られた反例からテストケースを生成する反例に基くテストケース生成機能 B，対象プログラムに過小近似の検知

用のトラップを挿入するカバレッジ用トラップ挿入機能 C，対象プログラムの DbC 仕様からテストケースを生成する仕様に基くテストケース生成機能 D，そして，生成されたテストケースを実行するテスト実行機能 F の 5 つの機能からなるツールセットとして実現される．本稿の時点では，機能 A と C を実装している．機能 B と D はそれぞれ既存研究 [20] と [62] をもとに実装可能である．機能 F はテストケースをテストプログラムとして実行し，その終了ステータスや出力されたログを調べるものであり，シェルスクリプトなどを用いて実装可能である．

図 6.5 においてツールセットが行うアルゴリズム `doSWMCplusTEST` を図 6.6 に示す．`doSWMCplusTEST` の終了ステータスは $true$ と $false$ の 2 値である．$true$ の場合，図 6.5 の一連の処理は終了である．$false$ の場合，図 6.5 の一連の処理は途中の可能性がある．ツールセットの利用者はソフトウェアモデル検査あるいはテストの結果を調べて，必要に応じて `doSWMCplusTEST` を再実行する．変数 `mode` は，`doSWMCplusTEST` が前回の実行時に設定した値を保持する永続データであり，その初期値は `SWMC` である．`doSWMCplusTEST` とは別に，`mode` の値を変更する機能があるものとする．`doSWMCplusTEST` は，大きく 4 つのパートに分かれている．ソフトウェアモデル検査を行うパート (行 1–7)，ソフトウェアモデル検査の結果に過大近似導入の影響があるかどうかを検知するパート (行 8–14)，ソフトウェアモデル検査の結果に過小近似導入の影響があるかどうかを検知するパート (行 15–22)，そして関数仕様に基づいたテストを行うパート (行 23–29) である．

まず，`doSWMCplusTEST` は，`mode` の値が `SWMC` の場合，機能 A によって，ソフトウェアモデル検査を行う (行 2)．指定時間内に探索が終わらず，不具合の有無を判定できない場合は，テストのパートへ進むように `mode` に値 `TEST` を設定する (行 3)．探索が終わり，反例が一つもない場合は，過小近似の検知パートへ進むように，`mode` に値 `UA` を設定する (行 4)．探索が終わり，反例が一つでもある場合は，過大近似の検知パートへ進むように，`mode` に値 `OA` を設定する (行 5)．

次に，`doSWMCplusTEST` は，`mode` の値が `OA` である，すなわち，モデル検査の結果が仕様違反の反例を含む場合は，機能 B によって，反例に基づいてテストケースを生成する (行 9)．モデル検査の反例からテストケースを生成することはよく知られており，反例の初期状態において変数に割り当てられた値をテストの入力データとする [2]．機能 E

```
        Initial state : mode = SWMC
    doSWMCplusTEST( program )
 1.  if ( mode = SWMC ) then /* Software Model-Checking */
 2.      swmcResultSet ← doSWMC( program )
 3.      if ( inconclusive( swmcResultSet ) ) then mode ← TEST
 4.      else if ( noCounterExamples( swmcResultSet ) ) then mode ← UA
 5.      else mode ← OA /* Counter-Examples exist */
 6.      endif
 7.  endif
 8.  if ( mode = OA ) then /* Detection of Over-Approximation */
 9.      testCaseSet ← doTCGfromCE( swmcResultSet, program )
10.      testResultSet ← doTest( testCaseSet, program )
11.      if ( reproducible( testResultSet ) ) then mode ← SWMC ; exit( true )
12.      else mode ← SWMC ; exit( false ) /* Re-run is needed */
13.      endif
14.  endif
15.  if ( mode = UA ) then /* Detection of Under-Approximation */
16.      trapProgram ← instrumentTrap( program )
17.      swmcResultSet ← doSWMC( trapProgram )
18.      if ( inconclusive( swmcResultSet ) ) then mode ← TEST
19.      else if ( unexplored( swmcResultSet ) ) then mode ← TEST
20.      else mode ← SWMC ; exit( true )
21.      endif
22.  endif
23.  if ( mode = TEST ) then /* Auxiliary Testing */
24.      testCaseSet ← doTCGfromSpec( program )
25.      testResultSet ← doTest( testCaseSet, program )
26.      if ( covered( testResultSet ) ) then mode ← SWMC ; exit( true )
27.      else exit( false ) /* Re-run might be needed */
28.      endif
29.  endif
```

図 6.6　有界モデル検査をテストケース生成で補うアルゴリズム

によって，生成されたテストケースを実行して，不具合が再現すれば，一連の処理を終了する (行 11)．利用者は，不具合を除去するように対象プログラムを修正する必要がある．不具合が再現しなければ，反例は過大近似の影響による誤警告であることがわかる．`doSWMCplusTEST` は，再実行時にソフトウェアモデル検査を行うように，`mode` に値 `SWQC` を設定し，終了する (行 12)．利用者は，ボディのソースコードをツールセットに与えて

いない関数について事後条件を追記する，あるいは，ソフトウェアモデル検査機能 A に与えるコマンドオプションに関して，変数として扱う配列要素数とループボディの探索回数とを整合するように決めるといった，過大近似の影響による誤警告を排除するための作業を行い，doSWMCplusTEST を再実行する．

さらに，doSWMCplusTEST は，mode の値が UA である場合，過小近似の影響があるかどうかを調べる．機能 C によって，カバレッジ基準に則って決めた箇所が探索された場合に真になるトラップ変数と，トラップ変数が偽であるというトラップ表明とを，対象プログラムに挿入し，トラップ入りプログラムを生成する (行 16)．機能 A によって，トラップ入りプログラムに対してモデル検査を行い，トラップ表明違反の反例を得る (行 17)．この反例は，トラップ変数が真になること，すなわちカバレッジ基準に則って決めた箇所をモデル検査によって探索したことを示す．指定時間内に探索が終わらず，トラップ表明についての検査結果を得られない場合は，テストのパートへ進むように mode に値 TEST を設定する (行 18)．探索が終わり，一つでも反例の無いトラップ表明があれば，未探索箇所があるものとして，テストのパートへ進むように mode に値 TEST を設定する (行 19)．探索が終わり，全てのトラップ表明に反例があれば，未探索箇所がないものとして，一連の処理を終了する (行 20)．

最後に，doSWMCplusTEST は，mode の値が TEST である場合，DbC 仕様に基づき対象プログラムをテストする．機能 D によって DbC 仕様からテストケースを生成し，機能 E によって生成されたテストケースを実行する (行 24–25)．テスト結果を調べて，カバレッジ基準を満たす場合は，一連の処理を終了する (行 26)．カバレッジ基準を満たさない場合は，再実行時にテストを行うように，mode の値 (TEST) を変更せずに，終了する (行 27)．利用者は，カバレッジを増やすために DbC 仕様をさらに詳細に場合分けして，doSWMCplusTEST を再実行する．なお，事前条件を前提とするとコードを実行し得ないといった正当な理由がある場合は，doSWMCplusTEST を再実行する必要はない．この場合，利用者は別機能を用いて mode の値を SWMC に戻す．

以降の 6.2.2 節と 6.2.3 節では，モデル検査では探索されない箇所の検知と DbC 仕様からのテストケース生成について述べる．

6.2.2 近似導入の検知方法

```
    #define  SIZ  1024
    extern int a[SIZ];
    extern int g;
    /**
 1:  @invariant g==-1 || g==0
     */
    /**
 2:  @post  __return==0 || __return==1
     */
    int bar();
    /**
 3:  @pre  __exist( k, 0, SIZ-1, a[k]==v ) || \
 4:        __foreach( k, 0, SIZ-1, a[k]!=v )
 5:  @post 0<=__return && a[ __return ]==v || __return==-1
    */
    int foo( int v ){
 6:   int r=-1, i=0;
 7:   if ( bar()+g==1 ) {
 8:     r=-2; /* BUG */
 9:     while ( i<SIZ ) {
10:       if ( v==a[i] ) { r=i; break; }
11:       i=i+1;
        }
      }
12:   return r;
    }
```

図 6.7 過小近似の影響を受けるプログラム例

過小近似の導入による問題について，ループの繰返しを所与の回数に限定する近似を例として，図 6.7 を用いて説明する．プログラムの複雑さに応じて状態遷移は様々に変わるので，有界モデル検査の探索の深さとループの繰返し回数の上限との関係は一意には決まらない．ここでは，簡単のため所与の回数を 2 回とする．検査の対象関数 `foo` の DbC 仕様では，引数 `v` は大域配列 `a` のいずれかの要素と値が同じか (行 3)，全ての要素と値が異なる (行 4)．`foo` の戻り値は `0` 以上か`-1` であり，前者の場合は戻り値を添え字とする `a`

の要素が v と同じ値を持つ (行 5)．foo の関数ボディでは，他の関数 bar の戻り値と大域変数 g の値の和が 1 の場合に (行 7)，配列 a の各要素と引数 v の関係を調べるループを実行する (行 9–11)．引数 v と値の同じ要素があれば，その要素の添え字を変数 r に代入して返す (行 9,12)．それ以外の場合には，事後条件から考えると本来は r の初期値-1(行 6) を返すべきである．しかし，r には-2 が代入されており (行 8)，配列要素と引数の値が一致しない場合には，この-2 が戻り値となり，事後条件への違反となる．過小近似されたプログラムは，ループボディ (行 10,11) の 3 回目の実行を行う前に処理を終了する．過小近似されたプログラムをモデル検査で検証すると，前述の事後条件違反の不具合は検出されない．しかし，ループボディの 3 回目以降については，モデル検査では調べておらず，事後条件が守られたのかどうかは判らない．

プログラムの経路は無数にありえるので，探索されない経路を探し尽くすことは難しい．経路の代わりに，プログラムの特定の箇所が探索されないことを検知する．この検知は，特定の箇所でのみ真になるトラップ変数を埋め込み，対象プログラムの終了箇所でトラップ変数が偽であるというプロパティをモデル検査で調べればよい．反例がなければ，特定の箇所は探索されていない．探索されない箇所というプログラム内部の性質について調べるので，プログラム構造に着目したカバレッジ基準に則って，特定の箇所を決める．カバレッジ基準としては，CACC (Correlated Active Clause Coverage) [3] を用いる．CACC は産業界で実用的に使われている masking MCDC に相当する．分岐文やループ文における制御式の全体を述語と呼ぶ．述語は二項論理演算子 ($\vee, \wedge$) で結合された節からなり，節は二項論理演算子を含まない．着目した 1 つの節を主節と呼び，残りの節を副節と呼ぶ．ある述語についてのテストが CACC を満たすためには，その述語の任意の主節について次の 2 点が要求される．

TR1　主節は真と偽のいずれにも評価されることがある．

TR2　主節のある値について述語を真にするような副節の値の組があり，主節のもう一方の値について述語を偽にするような副節の値の組がある．

なお，TR2 において副節の値の組は異なっていてもよい．

表 6.1　トラップ挿入のための整形ルール

rule	before	after

A1	while (P) {
	S
	}
	(k **番目のループ**)
		while (1) {
		if (P) { ; }
		else { goto break_k; }
		S
		continue_k:
		}
		break_k:

A2	do {
	S
	} while (P);
	(k **番目のループ**)
		while (1) {
		S
		continue_k:
		if (P) { ; }
		else { goto break_k; }
		}
		break_k:

A3	for (I;P;X) {
	S
	}
	(k **番目のループ**)
		I;
		while (1) {
		if (P) { ; }
		else { goto break_k; }
		S
		continue_k: ;
		X;
		}
		break_k: ;

A4	continue; (k **番目のループ内**)	goto continue_k;
A5	break; (k **番目のループ内**)	goto break_k;

B1	switch (X) {
	case C:
	S
	(**選択肢分繰り返し**)
	default:
	T
	}
		if (X==C) {
		goto case_k_C;
		(**選択肢分繰り返し**)
		}
		goto case_k_default;
		if (0) {
		case_k_C:
		S
		(**選択肢分繰り返し**)
		case_k_default:
		T
		}

B2	`v = P ? Q : R;`	`if (P) {`
		`  v = Q;`
		`} else {`
		`  v = R;`
		`}`

C	`if (P) {`
	`  S`
	`}`
		`if (P) { pred=1; }`
		`else  { pred=0; }`
		`if (pred) {`
		`  S`
		`}`

D1	`if (!P) { pred=b1; S }` `else  { pred=b2; T }`	`if (P) { pred=b2; T }` `else  { pred=b1; S }`

D2	`if (P && Q) { pred=b1; S }`
	`else  { pred=b2; T }`
		`if (P) {`
		`  if (Q) { pred=b1; S }`
		`  else  { pred=b2; T }`
		`}`
		`else  { pred=b2; T }`

D3	`if (P || Q) { pred=b1; S }`
	`else  { pred=b2; T }`
		`if (P)  { pred=b1; S }`
		`else {`
		`  if (Q) { pred=b1; S }`
		`  else  { pred=b2; T }`
		`}`

D4	`if (P ? Q : R) { pred=b1; S }`
	`else  { pred=b2; T }`
		`if (P)  {`
		`  if (Q) { pred=b1; S }`
		`  else  { pred=b2; T }`
		`} else {`
		`  if (R) { pred=b1; S }`
		`  else  { pred=b2; T }`
		`}`

E1	`if (P) { pred=b; }`	`if (P) { pred=b; trap_i_j=1; }`
E2	`else  { pred=b; }`	`else  { pred=b; trap_i_j=1; }`

F	`return` **<戻り値>**`;`	`__assert(trap_i_j==0);`
		（トラップ変数分，繰り返す）
		`return` **<戻り値>**`;`

`!`:否定．`&&`:論理積．`||`:論理和．
`P,Q`:述語．`S,T`:空ではない文やブロックの並び．`I,X`:式．`C`:定数．
`k`:関数内でのループあるいは switch の連番．
`pred`:述語の値を保持する変数．`b,b1,b2`:真偽値（`b1`≠`b2`）．
`trap_i_j`:トラップ変数，i は関数ごとの述語あるいは switch 文の連番，j は i ごとの節の組合せの連番．

カバレッジ基準に則ったトラップを埋め込むために，表 6.1 の整形ルールを用いて，プ

ログラムを整形し，トラップ入りプログラムを得る．

整形では，まず，ルール An(n=1,⋯) によって，ループ (`while`，`do`，`for`) を，ループの実行前後で変数値が同じになるという意味で，等価な `while` 文に変換する．変換後のループは，ループの制御式の評価をループボディ内で `if` 文を用いて行う．多分岐 (`switch`) と 3 項演算子 (`? :`) は，ルール Bn(n=1,⋯) によって同様の処理結果をもたらす `if` 文を用いた表現に変換する．

次に，ルール C によって，制御式 (述語) の値を保持する変数 `pred` を導入し，元のループや分岐全体を，`pred` に値を設定する文の集まりと，`pred` の値に応じて元のループボディや副文を行う文の集まりに分ける．

ルール Dn(n=1,⋯) は，`pred` に値を設定する文の集まりに対する変換である．`pred` に述語の値を設定する分岐を，述語を節に分解した入れ子の分岐に変換する．変換後の分岐全体において，ある分岐に着目すると，その分岐の節の値によって，述語 `pred` に異なる真偽値 (表 6.1 の b1,b2) を設定する代入文をもつ末端のブロックが存在するので，CACC の要求 TR1 を満たしえる．また，そのようなブロックへの経路を考えると，他の節の値の組を決めることができるので，CACC の要求 TR2 を満たしえる．すなわち，モデル検査によって末端のブロックを全て探索すれば，CACC を満たす検査を行えたといえる．

さらに，ルール En(n=1,...) によって，分岐の末端のブロックにトラップ変数 `trap_i_j`(i=1,⋯. j=1,⋯) を導入し，ブロックが探索されたことを示す値として真 (`1`) を設定する（トラップ変数は偽 (`0`) で初期化する）．

最後に，ルール F によって，`return` 文の直前に，トラップ変数ごとに，その値が偽であるというトラップ表明`__assert(trap_i_j==0);` を挿入する．モデル検査によって見つかったトラップ表明違反の反例は，トラップ変数に真を設定する箇所が探索されたことを示す．トラップ表明の全てについて反例が見つかれば，CACC を満たす探索が行われており，一つでも反例が見つからなければ，探索されない箇所があることが検知されたと結論づけてよい．

図 6.8 は，図 6.7 のプログラムを，表 6.1 のルールを用いて整形した例である．図 6.7 の 2 箇所の `if` 文 (行 7,10) と `while` 文 (行 9) は，それぞれ図 6.8 の行 7a–c と 10a–c および 9a–g に整形される．`if` および `while` 文の各制御式の真偽が決まる箇所には，カバ

```
     int foo( int v ){
       int pred;
       int trap_1_1=0, trap_1_2=0;
       int trap_2_1=0, trap_2_2=0;
       int trap_3_1=0, trap_3_2=0;
 6 :   int r=-1, i=0;
 7a:   if ( bar()+g==1 ) { pred=1; trap_1_1=1; }
 7b:   else              { pred=0; trap_1_2=1; }
 7c:   if ( pred ) {
 8 :     r=-2; /* BUG */
 9a:     while ( 1 ) {
 9b:       if ( i<SIZ ) { pred=1; trap_2_1=1; }
 9c:       else         { pred=0; trap_2_2=1; }
 9d:       if ( pred ) { ; }
 9e:       else { goto break_1; }
10a:       if ( v==a[i] ) { pred=1; trap_3_1=1; }
10b:       else           { pred=0; trap_3_2=1; }
10c:       if ( pred ) { r=i; goto break_1; }
11 :       i=i+1;
 9f:     continue_1: ;
   :     }
 9g:     break_1: ;
   :   }
12a:   __assert( trap_1_1==0 );
12b:   __assert( trap_1_2==0 );
12c:   __assert( trap_2_1==0 );
12d:   __assert( trap_2_2==0 );
12e:   __assert( trap_3_1==0 );
12f:   __assert( trap_3_2==0 );
12 :   return r;
   : }
```

図 6.8　未探箇所を検知するためのトラップ入りプログラムの例

レッジを示すためにトラップ変数に真 (1) を代入する文が挿入される (行 7a–b，10a–b，9b–c)．また，トラップ変数が偽 (0) であるというトラップ表明が `return` 文の直前に挿入される (行 12a–f)．

トラップ入りプログラム（図 6.8）に対するモデル検査のカバレッジを表 6.2 に示す．見出し行の「i,j」はトラップ変数 `trap_i_j` に対応しており，SWMC 行の各セルの Y は

表 6.2　モデル検査によるカバレッジの例

	1,1	1,2	2,1	2,2	3,1	3,2
SWMC	Y	Y	Y		Y	Y

見出し i,j： i 番目の述語における節の値の j 番目の組合せ

トラップ変数に真 (1) を代入する箇所がモデル検査で探索されたことを示す．ループボディを 2 回のみ実行する近似の導入により，`while` 文の制御式 (`i<SIZ`) が偽となる場合（図 6.8 行 9c）が探索されていないことが分かる．

6.2.3　仕様からのテストケース生成

ソフトウェアモデル検査で探索されない箇所が見つかった場合，互いに素な項からなる DNF 形式の DbC 仕様からテストケースを生成し，テストを行う [32]．ここで，項 A と B は $A \wedge B = false$ であれば，互いに素である．任意の論理式は DNF 形式に変換可能であり，さらに次の変換ルールを用いて互いに素な項からなる DNF 形式に変換できるので，一般性を失わない．

- $A \vee B = (A \wedge \neg B) \vee (A \wedge B) \vee (\neg A \wedge B)$
- $A \Rightarrow B = \neg A \vee (A \wedge B)$
- $A \Leftrightarrow B = (A \wedge B) \vee (\neg A \wedge \neg B)$

検査対象の関数について，その事前条件 Pre と事後条件 $Post$ が次の DNF 形式であるとする．

$Pre = \bigvee_i P_i$　　(DNF の各項は互いに素)

$Post = \bigvee_j (G_j \wedge D_j)$　　(DNF の各項は互いに素)

事後条件はガード条件 G_j と定義条件 D_j から構成されており，ガード条件は関数によって値が変更される変数を含まず，定義条件はそれらを含む．関数全体の振舞いは機能シナリオフォーム FSF で表される．

$FSF = \bigvee_{i,j} fs_{ij} = \bigvee_{i,j} (P_i \wedge G_j \wedge D_j)$

FSF 中の各 fs_{ij} を機能シナリオと呼ぶ．上記 FSF は，劉らが提案した FSF [62] に対して，事前条件も DNF 形式とし，機能シナリオをより細分化してある．機能シナリオ

fs_{ij} のうち $P_i \wedge G_j$ をテスト条件と呼び，テスト条件が恒偽でない場合に，テスト条件を満たす入力データを自動生成する．テストケースの自動生成では，対象関数の機能シナリオ以外に，対象関数によって参照・更新される大域変数の値域や，対象関数から呼ばれる関数の振舞いも考慮する必要がある．例えば，図 6.7 のプログラム例では，大域変数 g に関する不変条件 (行 1) と呼ばれる関数 bar の事後条件 (行 2) から，g と bar の取りうる値の組合せは (-1,0),(-1,1),(0,0),(0,1) の 4 通りであり，(0,1) の場合にのみ if 文の制御式 (bar()+g==1. 行 7) は真となる．分岐カバレッジを満たすためには，g の値と bar の戻り値を考慮する必要がある．そこで，大域変数の不変条件 $Inv = \bigvee_k I_k$ と，呼ばれる関数の事後条件 $Post^{called} = \bigvee_l Q_l$ も考慮して，$P_i \wedge G_j \wedge I_k \wedge Q_l$ を満足するように，対象関数への入力データと，呼ばれる関数の戻り値および副作用として変更される変数値とを算出し，D_j を期待結果とする．図 6.7 のプログラム例では，大域変数の不変条件は，g の値が-1 または 0 の 2 通り (行 1)，呼ばれる関数の事後条件も bar の戻り値が 0 または 1 の 2 通り (行 2)，対象関数の事前条件は，大域配列のいずれかの要素と引数の値が一致する場合 (行 3) と一致しない場合 (行 4) の 2 通りで，合計 8 通りの組合せがある．

図 6.9 は，図 6.7 のプログラム foo に関して，SMT ソルバを用いてテストケースを生成するためのスクリプト例である．機能シナリオの 1 つを満たす変数値の割当を，命題論理と整数の線形演算の範囲で解く．機能シナリオとしては，対象関数の事前条件 P_i として引数 v が大域配列 a のいずれの要素とも値が異なる場合を，大域変数の不変条件 I_k として g の値が 0 である場合を，呼ばれる関数の事後条件 Q_l として bar の戻り値が 1 である場合を選んでいる．対象関数の事後条件には，関数 foo によって更新されない変数が出現していないので，ガード条件 G_j は恒真であり，スクリプトでは省略している．話を簡単にするために，事前条件に出現する配列 a のサイズは-3 とする．配列の各要素 a[i](i=0,..) はスクリプトでは独立した変数 a__i としている．

```
> yices -f foo.smt
sat

MODEL
```

```
(benchmark  foo.smt
  :source { foo.c }
  :status unknown
  :category { sample }
  :logic QF_LIA          ; 整数の線形数値演算
; 大域変数の宣言
  :extrafuns (( a__0 Int ))  :extrafuns (( a__1 Int ))
  :extrafuns (( a__2 Int ))  :extrafuns (( a__3 Int ))
  :extrafuns (( g Int ))
; 呼ばれる関数の（戻り値）の宣言
  :extrafuns (( bar Int ))
; 引数の宣言
  :extrafuns (( v Int ))
; 機能シナリオ
  :assumption
  ( and
    ( = g 0 )            ; Ik : 不変条件
    ( = bar 1 )          ; Ql : 呼ばれる関数の事後条件
    ( and                ; Pi : 対象関数の事前条件
      ( not ( = a__0 v ) )
      ( not ( = a__1 v ) )
      ( not ( = a__2 v ) )
    )
  )
)
```

図 6.9 テストケース生成用スクリプトの例

```
(= g 0)
(= v 0)
(= a__1 1)
(= a__2 1)
(= bar 1)
(= a__0 -1)
----
```

図 6.9 のスクリプトを入力として SMT ソルバ (yices) を実行すると，上記のように変数値の割当が見つかる．

図 6.7 のプログラム foo に関して，DbC 仕様を元に図 6.9 のスクリプトを用いて

```
   /* Ql : post-conditions of called functions */
1: int bar() { return 1; }
   void test1(){
     int result, defining;  int v;
     /* Ik : invariants of global variables */
2:   g = 0;
     /* Pi ∧ Gj : pre-/guard-conditions of parameters */
3:   a[0] = -1;
4:   a[1] = 1;
5:   a[2] = 1;
6:   v=0;
7:   result = foo(v);
     /* Dj : defining conditions as expected outputs */
8:   defining = (0<=result && a[result]==v);
9:   if (defining) printf("OK"); else printf("NG");
   }
```

図 6.10　テストプログラムの例

生成したテストプログラム test1 を図 6.10 に示す．関数 bar については事後条件 (__return==1) を満たすスタブ関数を生成する (行 1)．大域変数 g には不変条件 (g==0) を満たす値を代入する (行 2)．大域配列 a の各要素と引数 v には対象関数の事前条件 (__foreach(k, 0, SIZ-1, a[k]!=v)) を満たす値を代入する (行 3–6)．テストプログラムは，生成した入力データを与えて対象プログラムを実行し (行 7)，実行結果について定義条件 (0<=__return && __return==v) の値を調べ (行 8)，ログを出力する (行 9)．元のプログラムの代わりにトラップ入りプログラム（図 6.8）を用いて，トラップ表明 (__assert) をログ出力に置き換えてカバレッジを調べた結果を表 6.3 に記す．行 $SWMC$ は表 6.2 に同じである．行 $Testing$ 列 i, j の Y は，トラップ変数 trap_i_j に真 (1) を代入する箇所がテスト実行されたことを示す．列 $2, 2$ を見ると，ソフトウェアモデル検査で探索しなかった箇所をテスト実行している．ソフトウェアモデル検査をテストで補完することにより，両者を合せて少なくとも CACC 以上の網羅度での検査が行えている．

表 6.3　モデル検査とテストによるカバレッジの例

	1,1	1,2	2,1	2,2	3,1	3,2
SWMC	Y	Y	Y		Y	Y
Testing	Y		Y	Y		Y

見出し i,j： i 番目の述語における節の値の j 番目の組合せ

6.3　実験

提案ツール (図 6.5) において，トラップ挿入ツールを作成し，ソフトウェアモデル検査ツール VARVEL と組み合わせて，オープンソースの OS である MINIX(Version 3.1.1) のソースコードの一部に適用した．MINIX は，ソースコードが公開されている点と，文献 [79] に仕様が記載されている点で，DbC の検査を行いやすい．MINIX は，マイクロカーネルのアーキテクチャを採用しており，サーバ，カーネル，ドライバ層の各プロセス間でのメッセージ通信による協調動作を特徴とする．ユーザアプリケーションからドライバへの要求は，サーバ層のファイルシステム機能を介して行われる．とくにファイルシステム機能のファイル `device.c` にはドライバとのメッセージ通信に関する関数が集められており，文献記載の MINIX の特徴を検査するのに適している．

`device.c` の主要な関数について，本稿の提案方法に則って，ソフトウェアモデル検査を行い，探索されない箇所があればテストケースを生成して，テストを行った．その結果を表 6.4 に示す．表 6.4 の列において，Function，Size，#Trap，#Total と#TC は，それぞれ関数名，関数の規模 (行数)，有界モデル検査によって探索されたトラップの個数，テスト実行されたトラップの個数，モデル検査とテスト実行のいずれかでカバーされたトラップの個数，そして，DbC 仕様に基づいて生成されたテストケースの個数である．列#Test と#TC の--は，テストケース生成およびテストを行わなかったことを示す．

DbC 仕様としては，メッセージのタイプは特定の整数値であること，プロセス番号は-4 以上かつ 100 未満であること，デバイス番号は 0 であるか，16 ビット中の上位 8 ビットが 32 未満であること，処理の結果を表す値は成功が 0 であり，失敗が負値であること，の 4 点を DNF 形式で記述した．モデル検査では，これらの DbC 仕様をプロパティとし

表 6.4　MINIX への提案方法の適用結果

Function	Size	#Trap	#BMC	#Test	#Total	#TC
dev_open	20	6	6	–	6	–
dev_close	9	2	2	–	2	–
dev_status	41	21	19	4	21	1
dev_io	48	8	8	–	8	–
tty_opcl	32	9	4	8	9	32
ctty_opcl	12	2	2	–	2	–
do_setsid	16	2	2	–	2	–
do_ioctl	39	5	5	–	5	–
gen_io	76	25	22	18	25	144
ctty_io	21	2	2	–	2	–
clone_opcl	52	7	7	–	7	–

Function：関数名.　Size：関数の空白行を除く行数.
#Trap：カバレッジ測定用のトラップ変数の個数.
#BMC：有界モデル検査 (BMC) で探索されたトラップの個数.
#Test：テスト実行されたトラップの個数.
#Total：BMC とテスト実行のいずれかでカバーされたトラップの個数.
#TC：生成されたテストケースの個数.

た．テストケースは，DNF 形式の DbC 仕様の各節から引数や大域変数の代表値の組を求め，これらの代表値の組の直積として得た．

ソフトウェアモデル検査には VARVEL(Version 3.5) を用いた．1 関数あたりの検査の上限時間と BMC の最大メモリ使用量として，それぞれ 60 分と 1G バイトを指定した．ループはボディを最大でも 2 回まで繰り返す設定とした．テストケースの各代表値は SMT ソルバ Yices [34] (Version 2) を用いて論理演算と線形整数演算の範囲で求めた．いずれも，CPU クロックは 2.53 GHz，メモリ容量は 4G バイトの Linux 機上で実行した．

表 6.4 の関数 dev_open では，6 個のトラップの全てをモデル検査で探索できたので，テストは実行していない．関数 dev_status では，21 個のトラップのうち 19 個をモデル検査によって探索した．また，残りの 2 個のトラップを含む 4 個のトラップを 1 個のテストケースによって実行した．dev_status の各トラップが有界モデルとテストにより，どのようにカバーされたかを表 6.5 に示す．見出し行の i, j はトラップ変数 trap_i_j に

対応する．行 $SWMC$ の Y はトラップがソフトウェアモデル検査によって探索されたことを，行 $Testing$ の Y はテスト実行されたことを示す．列 $1,1$ と $3,1$ を見ると，ソフトウェアモデル検査をテスト実行で補えたことがわかる．

表 6.5 MINIX を対象としたモデル検査とテスト実行によるカバレッジ

	1,1	1,2	2,1	2,2	2,3	3,1	3,2	4,1	4,2	5,1	5,2
SWMC		Y	Y	Y	Y		Y	Y	Y	Y	Y
Test	Y	Y		Y		Y					

	6,1	6,2	7,1	7,2	8,1	8,2	8,3	8,4	9,1	9,2	
SWMC	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
Test											

見出しの i,j： i 番目の述語における節の値の j 番目の組合せ

テストケースが十分ではなく，実行できないトラップがある場合には，プログラムコードを調べて，トラップに関係する DbC 仕様をより詳しく記述し，テストケースを作り直した．例えば，関数 `gen_io` では，呼ばれる関数`_sendrec` の戻り値が 0 以下であるという事後条件（`__return<=0`）を，戻り値の値を明示した DNF の式（`__return==0 || __return==-101 ||` $\cdots$）に修正した．テストケースの作り直しは，`gen_io` では，戻り値の場合分けが多く，人手の作業に約 1 時間かかったが，`tty_opcl` では約 10 分であった．修正後の DbC 仕様を用いても，表 6.4 のモデル検査で探索したトラップの数は変わらなかった．

表 6.4 では，全トラップ 89 個のうち，89% に当たる 79 個のトラップが有界モデル検査によって探索された，即ち，各トラップについて，トラップ変数に真を設定する箇所を通って関数の終了箇所に至る経路が網羅探索された．対象プログラムの大部分を有界モデル検査によって網羅的に検査し，残りの一部についても分岐カバレッジ基準 CACC を満たすテストを行っており，従来のテストのみの検査に比べてカバレッジの高い検査を実施できた．

引数や大域変数の値域の直積としてテストケースを生成する方法を用いて，テストのみで CACC 基準を満たそうとすると，関数 `gen_io` の場合，9418 個のテストケースが必要

であった．変数値の組合せを特定するように DbC 仕様を記述し，テストケースを減らすことができるが，プログラムの内部構造に合せて DbC 仕様を場合分けする必要があり手間がかかる．できるだけ多くの範囲を有界モデル検査で網羅的に検査し，一部についてのみテストケースを考慮する本稿のアプローチはテストの省力化という点で有効な方法である．

6.4　考察

有界モデル検査法 (BMC) を用いてプログラムを検査するツール [4] [25] [29] [54] [67] は不具合の検出に有効である．BMC を適用するために，プログラムを有限状態遷移システムに変換する際に，過小近似を導入することがある．ループを固定回数分のループボディの繰返しとするような変換を行うと [25] ，BMC では固定回数を超える部分を含む経路を探索しない．元のプログラムに置いて過小近似箇所を通る経路上の不具合を見逃す問題となる．

本章では，BMC を用いたソフトウェアモデル検査において，過小近似によって探索されない箇所が生じる問題を明らかにし，過小近似箇所の自動検知と DbC 仕様からのテストケース生成とを合せて，BMC を補完する方法を提案した．BMC における過小近似箇所検出と自動生成したテストケースを用いたテスト実行とで共通の分岐カバレッジ基準 CACC [3] （産業界における MCDC）を採用することにより，プログラム自動検証とプログラム単体テストとの関係を明らかにした．また，提案方法を MIINX の一部のソースコードに適用する実験を行い，ソフトウェアモデル検査とテスト実行を合わせて CACC を 100% 達成するという有効性を確認した．具体的な有界モデル検査ツールとしては VARVEL を用いたが，提案方法は CBMC といった C プログラムの有界モデル検査ツール一般に適用可能である．

今後の課題については，より大規模な実験が必要である．また，カバレッジ基準については，6.5 節に後述する PCT カバレッジのようなモデル検査の特性を考慮した基準が好ましく，検討の余地がある．

6.5　関連研究

有界モデル検査法 (BMC) を用いたソフトウェアモデル検査では，元のプログラムから有限状態遷移系への変換時に導入される過小近似によって，不具合の見逃しが起こる．探索範囲を制限しないように有界モデル検査を拡張する手法 [63] [68] が提案されているが，本章ではテスト技術との役割分担という方法を提案した．産業界では，信頼性の基準はテスト技術で与えられており，たとえ自動検証ツールを使った場合であっても，テスト技術による検査との関係を論じる必要があることが理由である．

プログラム単体テストでは，十分なテストを行ったことを，検査のカバレッジを測定して判断する．プログラム単体テストとプログラム自動検証の関係を論じる観点としてカバレッジは必須である．モデル検査におけるカバレッジとシミュレーションにおけるカバレッジの関係を論じた研究がある [21]．シミュレーションにおける分岐カバレッジは，モデル検査においては，有限状態マシンのミュータントが仕様を満たすとしても，無意味に満たしていないかというカバレッジ (vacuity coverage) に相当する．モデル検査によって仕様が満足されることが判ったとしても，分岐と無関係に満足されるのであれば検査結果は無意味であり，有限状態マシンか仕様のいずれかを修正する必要があるとする．本章では，分岐カバレッジ基準を考慮したトラップ入りプログラムを対象としてモデル検査を行うことにより，未探索の分岐を調べる．未探索の分岐はプログラムが DbC 仕様を満たす検査結果に寄与しないので，vacuity coverage を調べていることになる．しかし，本章では，BMC のためにプログラムを有界な有限状態遷移系に変換する際の過小近似は必須であるとし，有限状態マシンや仕様を修正するのではなく，元のプログラムに対してテスト実行を追加するという考え方を取る．

検査対象にトラップ変数を挿入して，意図した経路を示す反例をモデル検査によって得る考え方がある [20]．この考え方によって，有限状態空間上の所望の状態や遷移を探索したか否かを調べることができる．本章では，分岐の述語を節に分解し，節の真偽の組合せごとにトラップを挿入することにより，モデル検査が CACC を満たす経路を探索したことを調べる．

本章で採用した CACC は，航空業界や自動車業界において高い信頼性を必要とするソフトウェアに要求される分岐カバレッジ基準であり，従来のプログラム単体テストとの関係を示すには適切である．しかし，有限状態空間を網羅探索するモデル検査自体のカバレッジを，分岐カバレッジ基準で測ることが適切とは言い難い．パスカバレッジは経路が無数にありえるため，カバレッジ基準としては不適切である．関連研究として，PCT(Predicate Complete Testing) カバレッジ [5] が提案されている．対象プログラムの文の個数 M と述語の個数 n に対して，各文における述語の取りえる値の組を 1 つの状態とする．対象プログラムをブーリアンプログラムに変換し，ブーリアンプログラム上の経路を辿って各状態への到達性を調べて，到達可能な状態の総数をカバレッジの分母とする．状態の総数は最大でも $M \times 2^n$ なので，分母を有限に押さえることができる．モデル検査のカバレッジは，PCT カバレッジのように経路を考慮し，かつ母数が有限のカバレッジ基準を用いて測ることが望ましい．

テスト技術と静的解析技術とを統合した手法として，オンラインテスト (online testing, concolic testing) がある [76] [81] ．Concolic testing では，まずテスト実行を行い，次に記号実行などの静的手法を用いて，実行経路上の各分岐条件ごとに実行済みの遷移を通らないような事前条件を求めて，入力データを生成し，次のテスト実行を行う．この一連の処理を繰り返し，分岐カバレッジ基準を満足するテスト実行を試みる．テスト実行とテストケース生成を 1 つのアルゴリズムに閉じて行うことにより，無駄なテストケースを生成しない点が興味深い．しかし，一度実行した分岐を実行しないようにする条件を自動証明器を用いて算出しているため，プログラムが非線形演算を含む場合に，テストケースを生成できないことがある．

本章の提案方法は，テスト実行とテストケース生成を分けて行う点でオンラインテストとは異なり，オフラインテストに属する．ただし，既存のオフラインテストが，テストケース生成のためにモデル検査を利用するのに対して，提案方法はソフトウェアモデル検査が主体であり，検査しきれなかった部分を少数のテストケースでカバーする点が異なる．また，プログラムが非線形演算を含む場合には，DbC 仕様に基づくテストケース生成をランダムテストケース生成などに変更することにより，対応できる．

第 7 章

結論

7.1 成果

本稿では，有界モデル検査を用いたモジュラー検証について，実用化に向けた提案と，その実証を行った．

第 3 章では，有界モデル検査ツール VARVEL における関数仕様の記法と，仕様をプリミティブ関数へ変換することによるモジュラー検証の実現方法を述べ，産業界の C プログラムを対象とした実験により，ソフトウェアモデル検査をテスト作業の自動化技術と位置付けた．

第 4 章では，モジュラー検証の弱みである，大域情報の不足によって不具合の見逃ごしや誤警告が起こる問題に関して，再入の状況についての対策を講じた．ファイルをスコープとする不変条件の記法と，有界モデル検査ツールの特徴である関数呼出しのインライン展開によってコールシーケンス全体を一度に検査することを提案し，サンプルプログラムを対象とした実験により，再入箇所での不変条件違反をピンポイントで発見できることを示した．

第 5 章では，同じく大域情報の不足に起因する，関数ポインタを介した間接呼出しにおける誤警告に対して，解決を図った．関数ポインタ自体に DbC 仕様 (仮仕様) を付与する記法を提案した．また，モジュラー検証と仮仕様と間接的に呼ばれる関数の実仕様との一貫性検査の 2 段階の検査を行うアプローチを提案した．提案方法の効果について，MINIX を対象とした実験において，誤警告の削減および関数ポインタが関与する不具合

の発見を示した．

第 6 章では，ソフトウェアモデル検査を自動テストと位置付けるに当たり，プログラムの構造カバレッジの観点で従来の単体テストとの関係を論じた．ソフトウェアモデル検査の誤警告および不具合の見逃ごしの問題に関して，それぞれ自動検知の方法を提案し，後者については DbC 仕様からのテストケース生成によって，ソフトウェアモデル検査をテストで補う方法を提案した．第 5 章と同じく，MINIX を対象とした実験において，ソフトウェアモデル検査と，少数の自動生成テストケースによるテストとを合わせて，分岐カバレッジ基準を満たす検査を行えることを示した．

本研究で提案した有界モデル検査を用いたモジュラー検証は，事前・事後条件が入力データ・期待結果に相当することから，単体テストの自動実行手段と位置付けることができる．さらにソフトウェアモデル検査をテストケース自動生成で補う方法では，ソフトウェアモデル検査とプログラム・テストに共通のカバレッジ基準を用いて，検査が十分か否かを評価する．モジュラー検証においてプログラムの事前・事後条件を作成することは単体テスト計画に対応し，ソフトウェアモデル検査を自動生成テストで補う方法は単体テストの実施と評価を自動化することに相当する．本研究の提案方法は全体として，従来の単体テストの作業を自動化，効率化する．形式検証技術の産業界への移転を容易にする方法といえる．

7.2　今後の課題

モジュラー検証の実用化に向けては，機能・性能・使い勝手と様々な面でツールを強化する必要があるが，運用面においても関数仕様を書く工数をあまり増やさない工夫が必要となる．ロジック・モデル検査では，調べたい性質を時相論理式で表現する．全ての経路上の全ての状態や，ある経路上の状態 1 に続く状態 2 といった，経路および時間に跨る多様な表現が可能である．一方，モジュラー検証における DbC 仕様は，プログラムの特定の箇所で取得できる変数値を用いた局所的な条件である．時相論理における表現の一部分しか，あるいは全く，表現できないことがある．また，DbC 仕様の詳細度には，検査の精度とモデル検査の探索の性能に関するトレードオフがある．入力や出力の値域を記す

に留めると，粗い検査しかできないが，探索は早い．入力と出力の対応関係を細かく場合分けして記すと，探索は遅いが，きめ細かい検査ができる．プログラムの外部機能仕様の検査を目的とするのか，仕様に基づくテストケース生成を目的とするのか，目的に応じて記述の詳細度を考える必要がある．実用化に向けては，これらの課題に対して，DbC 仕様として記述できる・できない性質についての情報を提供し，プログラムの仕様をどこまで詳しく記述するかなどのノウハウを蓄積する必要がある．また，既にテスト済みのレガシーコードを呼び出すプログラムを開発する場合，レガシーコードに対して DbC 仕様を記述する必要がある．しかし，レガシーコードの仕様は明らかではないことがある．このような場合に，プログラムのコード自体から DbC 仕様を生成する研究がある [43]．本稿で用いた VARVEL についても，モジュラー検証への利用を視野に入れて，別プロジェクト [53] にて事前条件の自動生成機能を開発した．本研究が実用的なプロジェクトの成果への基礎を与えている．

謝辞

本研究を進め，本論文をまとめるにあたり，始終暖かい御助言と貴重なるご指導を賜り，また多くの御支援を頂戴いたしました国立情報学研究所 中島震教授に心より感謝の意を表します.

本論文をまとめるにあたり，有益な議論と貴重なる御助言を賜りました法政大学情報科学部 劉少英教授，大阪大学大学院情報科学研究科 岡野浩三准教授，国立情報学研究所細部博史准教授ならびに日高宗一郎助教に深く感謝の意を表します．国立情報学研究所 Franz Weitl 氏には有益な議論と関連文献のご紹介を賜りました．感謝の意を表します.

また本研究の機会を与えて下さいました日本電気株式会社 岩崎新一ソフトウェア生産革新部長，中央研究所の中田登志之主席技術主幹，高島洋典支配人，野口誠サービスプラットフォーム研究所所長，ならびに，日本電気株式会社知的資産 R&D ユニット歴代幹部の方々に心から感謝します.

NEC Laboratories America, Inc. の Aarti Gupta 氏率いる Systems Analysis & Verification グループからは，有界モデル検査ツール F-Soft をご提供いただき感謝します．とくに Franjo Ivanćiĉ 氏には SAT およびソフトウェアモデル検査に関する知見を教えていただき，心から感謝します．コロラド大学に移られた Sriram Sankaranarayanan 氏にはプログラム解析の知見をお教えいただき感謝します.

日本電気株式会社ソフトウェア生産革新部の皆様，とくに三橋二彩子グループマネージャ率いる検証技術グループからは，本研究で中心的な役割を担う有界モデル検査ツール VARVEL をご提供いただき感謝します．三橋氏には投稿論文を詳細にご確認いただいたことも感謝します．向山輝マネージャには SAT の概要やモデル検査によるテストケース生成の可能性を教えていただき感謝します．井元崇主任には VARVEL の使い方を詳しく

教えていただき感謝します．前田直人主任にはプログラム解析の知見をお教えいただき感謝します．同部の池田健次郎エキスパートには，DbC の基本機能に関する実験を中国にて共に遂行いただきました．また，モデル検査のカバレッジ測定についても有益なご意見をいただき感謝します．

最後に，いつも明るく支え続けてくれる妻 明美に感謝します．

参考文献

[1] Jean-Raymond Abrial. Formal methods in industry: Achievements, problems, future. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 761–768, 2006.

[2] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*, ICFEM 1998, pp. 46–54, 1998.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International journal on Software Tools for Technology Transfer (STTT)*, Vol. 11, pp. 69–83, January 2009.

[5] Thomas Ball. A theory of predicate-complete test coverage and generation. Technical report, Microsoft Research Technical Report, MSR-TR-2004-28, 2004.

[6] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Communications of the ACM*, Vol. 54, pp. 68–76, July 2011.

[7] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, PLDI' 01, pp. 203–213, 2001.

[8] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pp. 103–122, 2001.

[9] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT*

symposium on Principles of Programming Languages, POPL '02, pp. 1–3, 2002.

[10] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, MPC 2004, pp. 54–84, 2004.

[11] Mike Barnett. Code contracts for .NET: Runtime verification and so much more. In *Proceedings of the 1st international conference on Runtime Verification*, RV'10, pp. 16–17, 2010.

[12] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the 1st international workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, CASSIS 2004, pp. 49–69, 2004.

[13] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language.

[14] Boris Beizer. ソフトウェアテスト技法. 日経 BP 出版センター, 1994. 小野 間彰, 山浦 恒央 訳, (原書: Software Testing Techniques).

[15] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pp. 326–335, 2004.

[16] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International journal on Software Tools for Technology Transfer (STTT)*, Vol. 9, No. 5-6, pp. 505–525, 2007.

[17] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, Vol. 58, pp. 118–149, 2003.

[18] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th international conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pp. 193–207, 1999.

[19] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[20] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings of the 2nd international SPIN workshop on model checking of software*, SPIN '96, 1996.

[21] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for formal verification. *International journal on Software Tools for Technology Transfer (STTT)*, Vol. 8, No. 4–5, pp. 373–386, 2006.

[22] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM TOPLAS*, Vol. 16, No. 5, pp. 1512–1542, 1994.

[23] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, 1999.

[24] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th international conference on Computer Aided Verification*, CAV 2000, pp. 154–169, 2000.

[25] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th international conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '04, pp. 168–176, 2004.

[26] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pp. 368–371, 2003.

[27] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd international conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pp. 23–42, 2009.

[28] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '06, pp. 415–426, 2006.

[29] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Proceedings of the 2009 IEEE/ACM international conference on Automated Software Engineering*, ASE '09, pp. 137–148, 2009.

[30] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proceedings of the 27th ACM/IEEE International Conference on Software Engineering*, ICSE '05, pp. 422–431, 2005.

[31] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM (JACM)*, Vol. 52, No. 3, pp. 365–473,

2005.

[32] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the 1st international symposium of Formal Methods Europe on industrial-strength formal methods*, FME '93, pp. 268–284, 1993.

[33] Edsger W. Dijkstra. *A Discipline of Programming.* Series in Automatic Computation. Prentice Hall, 1976.

[34] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV 2006, pp. 81–94, 2006.

[35] Manuel Fähndrich, Diego Garbervetsky, and Wolfram Schulte. A re-entrancy analysis for object oriented programs. In *Proceedings of the 9th sorkshop on Formal Techniques for Java-like Programs*, FTfJP 2007 at ECOOP, 2007.

[36] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Proceedings of the 6th IEEE International Conference on Formal Engineering Methods*, ICFEM 2004, pp. 15–29, 2004.

[37] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pp. 48–59, 2002.

[38] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming Language Design and Implementation*, PLDI '02, pp. 234–245, 2002.

[39] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Proceedings of the 16th international conference on Computer Aided Verification*, CAV 2004, pp. 175–188, 2004.

[40] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '97, pp. 174–186, 1997.

[41] Patrice Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, Vol. 26, pp. 77–101, March 2005.

[42] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th international conference on Computer Aided Verification*, CAV '97, pp. 72–83, 1997.

[43] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 232–241, 2006.

[44] Yuusuke Hashimoto and Shin Nakajima. Modular checking with model checking. *ENTCS*, Vol. 254, pp. 105–122, October 2009.

[45] Yuusuke Hashimoto and Shin Nakajima. Modular checking of C programs using SAT-based bounded model checker. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pp. 515–522, 2009.

[46] 橋本祐介, 中島震. 有界モデル検査法を用いた C プログラムのモジュラー検証. 情報処理学会論文誌, Vol. 52, No. 8, pp. 2422–2430, 2011.

[47] 橋本祐介, 中島震. ソフトウェアモデル検査とテストケース生成の統合ツールチェイン. ソフトウェアエンジニアリングシンポジウム 2011 論文集, SES2011, 2011.

[48] 橋本祐介, 中島震. ソフトウェアモデル検査とテストケース生成の統合. 情報処理学会論文誌, Vol. 53, No. 2, pp. 548–556, 2012.

[49] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *International journal on Software Tools for Technology Transfer (STTT)*, Vol. 2, No. 4, pp. 366–381, 2000.

[50] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th international conference on Computer Aided Verification*, CAV 2002, pp. 526–538, 2002.

[51] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '02, pp. 58–70, 2002.

[52] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[53] Franjo Ivančić, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Mustafa Hussain, Naoto Maeda, Hiroki Tokuoka, Takashi Imoto, and Yoshiaki Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *Proceedings of the 26th IEEE/ACM international conference on Automated Software Engineering*, ASE 2011, 2011 (to appear).

[54] Franjo Ivančić, Ilya Shlyakhter, Aarti Gupta, Malay K. Ganai, Vineet Kahlon, Chao Wang, and Zijiang Yang. Model checking C programs using F-Soft. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pp. 297–308, 2005.

[55] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, Vol. 41, No. 4, pp. 21:1–21:54, 2009.

[56] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, Vol. 163, pp. 203–243, November 2000.

[57] Moonzoo Kim, Yunho Kim, and Hotae Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Transactions on Software Engineering*, Vol. 37, pp. 146–160, March 2011.

[58] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph Wintersteiger. Termination analysis with compositional transition invariants. In *Proceedings of the 22th international conference on Computer Aided Verification*, CAV 2010, pp. 89–103, 2010.

[59] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[60] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual.

[61] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6, pp. 1811–1841, November 1994.

[62] Shaoying Liu and Shin Nakajima. A decompositional approach to automatic test case generation based on formal specifications. In *Proceedings of the 4th international conference on Secure Software Integration and Reliability Improvement*, SSIRI 2010, pp. 147–155, 2010.

[63] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the 15th international conference on Computer Aided Verification*, CAV 2003, pp. 1–13, 2003.

[64] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, Vol. 25, No. 10, pp. 40–51, 1992.

[65] Bertrand Meyer. *The Dependent Delegate Dilemma*. Engineering Theories of Software Intensive Systems, Springer-Verlag, 2005.

[66] Bertrand Meyer. オブジェクト指向入門 第 2 版 原則・コンセプト. 翔泳社, 2007. 酒匂 寛 訳, (原書: Object-Oriented Software Construction (2nd Edition)).

[67] 宮崎義昭, 橋本祐介. C 言語へのフォーマルメソッドの適用. 情報処理, Vol. 49, No. 5, pp. 514–520, 2008.

[68] Leonardo De Moura, Harald Ruess, and Maria Sorea. Bounded model checking

and induction: From refutation to verification. In *Proceedings of the 15th international conference on Computer Aided Verification*, CAV 2003, pp. 14–26, 2003.

[69] 中島震. ソフトウェア工学の道具としての形式手法. ソフトウェアエンジニアリング最前線 2007 (改訂版：NII Technical Report 2007-007J), 2007 年 8 月.

[70] 中島震. ソフトウェア工学からみたモデル検査法. 第 22 回 回路とシステム軽井沢ワークショップ, 2009.

[71] 中島震. 「形式手法の産業界応用」を進めるには？ 形式手法の産業界応用ワークショップ 2011 予稿集, WIAFM2011, pp. 27–29, 2011.

[72] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th international conference on Compiler Construction*, CC '02, pp. 213–228, 2002.

[73] Pieter Philippaerts, Frédéric Vogels, Jan Smans, Bart Jacobs, and Frank Piessens. The Belgian electronic identity card: a verification case study. In *Proceedings of the 11th international workshop on Automated Verification of Critical Systems*, AVoCS '11, 2011.

[74] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International journal on Software Tools for Technology Transfer (STTT)*, Vol. 7, No. 2, pp. 156–173, 2005.

[75] Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the 10th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '04, pp. 404–420, 2004.

[76] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering*, ESEC/FSE-13, pp. 263–272, 2005.

[77] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '96, pp. 32–41, 1996.

[78] 玉井哲雄. ソフトウェア工学の基礎. 岩波書店, 2004.

[79] Andrew S. Tanenbaum. オペレーティングシステム 第 3 版. ピアソンエデュケーション, 2007. 吉澤 康文，木村 信二，永見 明久，峯 博史 訳，(原書: Operating Systems Design and Implementation (3rd Edition)).

[80] 梅村晃広．SAT ソルバ・SMT ソルバの技術と応用．コンピュータソフトウェア, Vol. 27, No. 3, pp. 24–35, 2010.

[81] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pp. 39–76. Springer-Verlag, 2008.

[82] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, Vol. 29, pp. 97–107, July 2004.