博 士 論 文

論理デバッグ・**ECO** のための
自動修正技術に関する研究
Automatic Rectification Methods for Logic Debugging and ECO

令和 1 年 11 月 29 日提出

指導教員　藤田 昌宏　教授

東京大学大学院工学系研究科
電気系工学専攻
37-177071
木村 悠介

概要 Abstract

The design of digital circuits may go through different abstraction levels like high-level (e.g. C-based) design, RTL design, gate-level design and so on. Sometimes, a logical bug-fix or engineering change order (ECO) may happen in the later phases of the design, and those changes need to be reflected in all the design descriptions of the target design. This research mainly deals with two situations: (1) C or RTL design is modified, and ECO should be done at gate-level in an automatical manner, (2) gate-level circuits are modified directly, and high-level C descriptions remains unchanged.

In the first situation, the process to modify gate-level circuits generally consists of three steps: (1) Finding the target gates to be modified, (2) Selecting internal signals for correcting the gates, (3) Generating new logic for fan-ins of the target gates (patch). My proposed methods focus on step (2). This research direction is common recently and essentially the same as ICCAD’17 contest. I propose a new formulation to solve multiple targets at the same time. It gets 10-100 times better results in terms of the contest criteria compared to the existing research.

Moreover, the proposed method can be applied to sequential circuits, by enumerating all the reachable states. The method can find correct solutions within practical time for all ITC99 benchmark circuits.

In the second situation, C-based design is assumed. The C description should be modified to become equivalent to the modified gate-level circuit. In this research, it is assumed that designers know which line(s) to be modified in the C description and have rough idea about how to modify them. Each target line is converted to a vacant place, and my proposed method synthesizes program expressions by utilizing the existing program synthesis techniques. It can rectify the C description of AES256 encryption successfully, and such a large program is not handled in the existing researches.

目次

第 1 章　序論　1

1.1　LSI の論理設計　2

1.2　デバッグ・ECO について　3

1.3　目的　10

1.4　本論文の構成　11

第 2 章　背景技術　13

2.1　SAT, QBF, ILP　14

2.2　論理回路の等価性検証　20

第 3 章　組合せ回路のための修正信号選択手法　23

3.1　既存研究　24

3.2　目的　25

3.3　問題の定義　25

3.4　複数ターゲットを同時に扱う定式化　27

3.5　ターゲット信号の優先順位　34

3.6　実験　36

3.7　結論　40

第 4 章　順序回路のための修正信号選択手法　42

4.1　既存研究　43

4.2　目的　43

4.3　問題の定義　44

4.4　例題　44

4.5　提案手法　49

4.6　実験　57

4.7 結論と今後の課題 66

第 5 章 修正されたゲートレベル回路からの C 記述再合成 69
5.1 既存研究 70
5.2 目的 71
5.3 問題の定義 71
5.4 提案手法 72
5.5 実験 80
5.6 結論 87

第 6 章 結論と今後の課題 89
6.1 結論 90
6.2 今後の課題 91

参考文献 95

表目次

2.1 Example of Set Covering 19

3.1 Finding an input for the patch 29
3.2 Finding inputs for the patch 30
3.3 ICCAD'17 contest circuits 37
3.4 Comparison between "At-the-same-time with ILP solver", "One-by-One (CSO) with ILP solver", and "One-by-One (CSO) with Heuristic solver" . . 39
3.5 Memory Usage (MB) 39
3.6 Comparison with the existing research 41

4.1 Reachable state pairs (Incorrect) 47
4.2 Reachable state pairs (Correct) 48
4.3 Statistics of ITC99 circuits 58
4.4 Buggy Circuits with Same Encoding (average of 10 problems) 60
4.5 Buggy Circuits with Different Encoding (average of 10 problems) 61
4.6 Memory Usage (MB) 62
4.7 Number of patch inputs (Same Encoding) 62
4.8 Number of patch inputs (Different Encoding) 62
4.9 Buggy Circuits with Different Encoding, with Data-path FF Correspondence (average of 10 problems) 63
4.10 Buggy Circuits with Wrong State Transition, with Data-path FF Correspondence 64

5.1 BitCount (Serial Formulation) 84
5.2 BitCount (Original Formulation) 84
5.3 Using different initial use-case (BitCount, Serial Formulation, ECO1). . . . 85

5.4 Using different initial use-case (BitCount, Original Formulation, ECO1). . . 85
5.5 AES implementation 85
5.6 AES (Serial formulation)) 87
5.7 AES (Original formulation)) 87

図目次

1.1 Common Design Flows . . . 2
1.2 Verification time ratio . . . 5
1.3 The number of bugs . . . 6
1.4 Reason of bugs . . . 8
1.5 Root cause of logic/functional bugs in ASIC development [1] . . . 9
1.6 Where verification engineers spend their time . . . 9
1.7 Automatic C modification based on a gatelevel circuit . . . 12

2.1 Miter . . . 20

3.1 Problem Definition . . . 26
3.2 Function(a) and Non-Function(b) . . . 27
3.3 Overall flow of the method for single target problems [2] . . . 28
3.4 Minimum Cut . . . 31
3.5 One-by-one method for multiple targets problems . . . 34
3.6 Importance of target priority . . . 35
3.7 Target priority and the execution time (X: time, Y: the number of target priorities) . . . 40

4.1 Retiming & Resynthesis . . . 43
4.2 State Correspondence (Moore-type FSM) . . . 43
4.3 Example circuit to be modified . . . 45
4.4 Spec circuit . . . 45
4.5 Spec logic . . . 45
4.6 Miter Circuit . . . 49
4.7 Overall flow of the proposed method . . . 50

4.8 Counter examples selection 51
4.9 Search for the reachable state pairs 54
4.10 Experiment3: two buggy state machines and the specification 65
4.11 Experiment3: Results 66
4.12 State correspondence: 1-1, 1-n, n-m 66
4.13 Minimizing Counter Example 67

5.1 General flow for C Reconstruction 73
5.2 A Block 75
5.3 3 Blocks 76
5.4 CEGIS flow 77
5.5 CEGIS flow for serial formulation 79
5.6 Fast algorithm for bitcount 83
5.7 AES calculation 86

第 1 章

序論

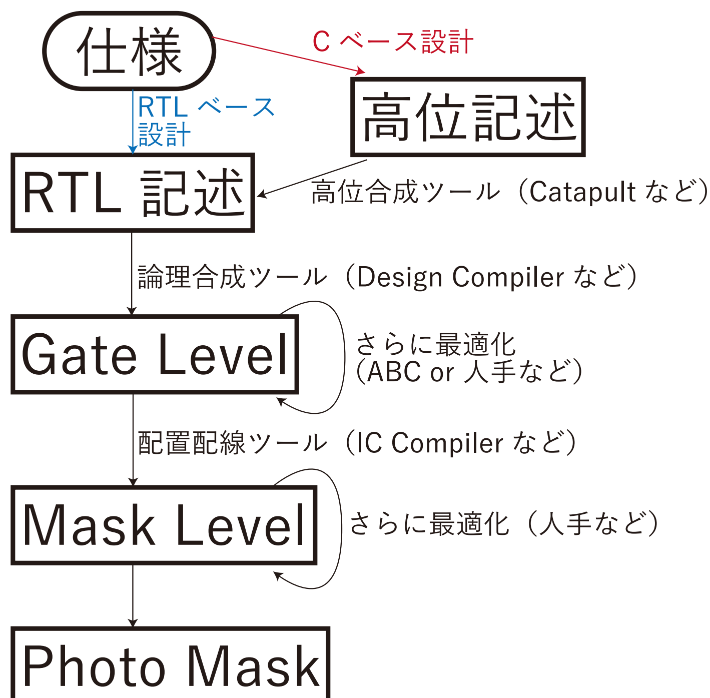


図 1.1: Common Design Flows

本章では、本論文に関する基本的な事項や、本研究の目的などについて説明する。

1.1　LSI の論理設計

LSI には数万から数億のトランジスタが搭載されることがあり、その設計をすべて手動で行うことは困難である。LSI 設計では古くから階層的な設計手法が用いられており、RTL ベース設計と C ベース設計という 2 つの設計フローが広く用いられている。以下にそれぞれの特徴を説明する。図 1.1 も参照されたい。

1.1.1　RTL ベース設計

LSI 設計で古くから用いられているのは､RTL ベース設計である。RTL(Register Transfer Level) とは、サイクルごとの動作を記述したものである。RTL ベース設計では、まず初めに RTL 記述を作成することから始める。大きな設計は順序回路として実装されることが多いから、設計者は実装したい機能の状態割り当てを考えて RTL 設計を書く。

RTL 記述は自動合成ツールによってゲートレベル設計に変換され、さらに配置配線が行われてマスクレベルのデータに変換される。自動生成されたデータはそのまま使われることもあるが、別のツールや人手で最適化されることも多い。最適化には、遅延・消費電

力・回路面積・デバッグのしやすさなどの様々な指標が使われる。

1.1.2　C ベース設計

C ベース設計は近年盛んに導入が進んでいる設計方法であり、C 言語などの高位言語から設計を始める点が特徴的である。前述の RTL ベース設計では、実装したい機能の状態割り当てを行ってサイクルごとの動作を設計者が決める必要があった。高位合成ツールによって C 言語から自動で RTL 記述を生成すれば、設計者は状態割り当てについて考える必要がなくなり、より簡単に LSI の論理設計が出来るようになる。RTL 記述が得られてからの設計フローは RTL ベース設計と同様である。

高位合成ツールは、最適な状態割り当てを提案したり、ソフトウェアで一般的な入出力（関数の入力値・返り値・標準入出力など）を自動でバスに変換したりする必要がある。その精度によって出来上がる LSI の性能が大きく異なってしまうから、実行速度が重要な回路の設計などでは依然として RTL ベース設計が行われている。逆に、FPGA の普及・機械学習やビックデータなどのアプリケーションの拡大によって、一般的な CPU よりも高速に演算できるデジタル回路に興味を持つソフトウェアエンジニアが増えており、そういった場面では高位合成ツールが広く活用されている。

1.2　デバッグ・ECO について

今日では、設計フローに沿って設計を行うだけでなく、設計の正しさも確認することが一般的である。「テスト・検証」と「デバッグ・ECO」に分けて概要を説明した上で、今日のチップ設計の現状について紹介する。

1.2.1　テスト・検証

LSI を設計する会社が設計途中で見つけたバグについて詳細なレポートをすることはないから、バグが見つからずに出荷されてしまったチップの例を見たい。1994 年に Intel の販売している Pentium プロセッサに、浮動小数点除算で誤った解が得られてしまうバグが発見されている [3]。CPU では除算時にはルックアップテーブルを活用して高速化を図る事が多いが、そのテーブルの一部が誤っていたことが原因である。浮動小数点演算の小さな誤りであったから、一般的なユーザーには大きな影響はなかったが、Intel は交換対応に 4 億ドルもの費用を見込んでいるとまで表明して事態を収めている。この事件後 Intel はマイクロコードの修正機構を導入しており、近年発見された Spectre [4] などの大規模バグでも「交換騒ぎ」にはなっていない。

以上のように、LSI では製造後のバグは深刻である。CPU のような幅広く用いられる高

価なチップならば OS と協調した修正機構が設けられるが、そのような例は少ない。従って、設計フローの途中や最後で適宜テスト・検証を行うことが一般的である。手間が増えてしまうが、マスク作成後や製造後にバグを見つけるよりは低コストであると考えられている。以下に代表的な手法を挙げる。

1. テストパターンを用いたテスト：いくつかの入出力パターンを用意しておき、正しい出力を得られるか検査する。記述中のアサーションが満たされるかどうかを調べることもある。
2. 形式検証：テストパターンを用意するのではなく、あらゆる入力について網羅的に調べる手法。特に 2 つのデザインの出力が等価になるか形式的に調べる手法は「等価性検証」と呼ばれて有名。アサーションについても形式検証することがある。

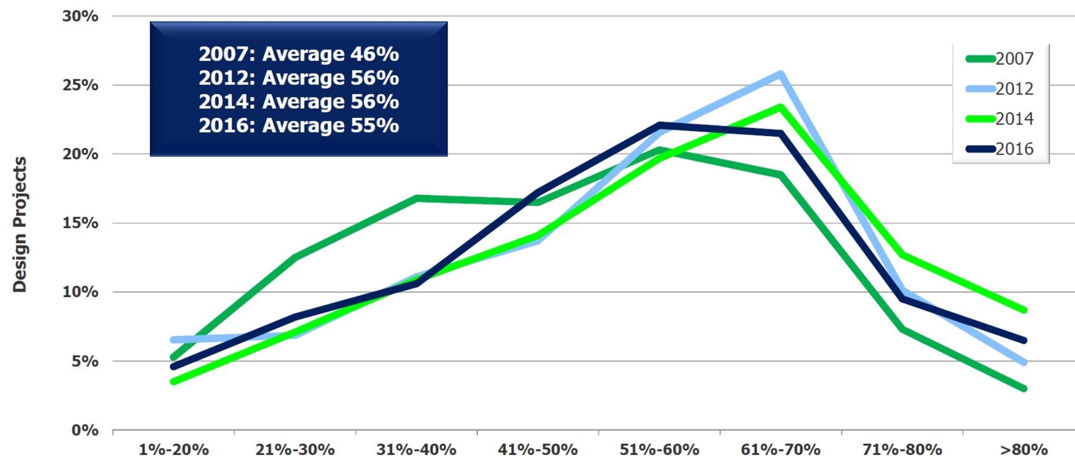
テストパターンを用いたテストは、ソフトウェア開発でも広く用いられている手法である。簡単に行える点が長所だが、与えられたテストパターンでの正しさしか分からないという点で網羅性が低い。等価性検証は設計を網羅的に検証出来るという点でテストパターンを用いたテストよりも高度だが、大きな設計データを扱えないという点が欠点である。多くの場合、システム全体ではなく設計データのモジュールごとなど、小さな単位で適用されることが多い。

設計現場に応じてどのテスト・検証手法を用いるかのポリシーは異なる。特に近年盛んな C ベースの FPGA 設計では、ツールが出力する回路データは C 記述と等価であると信じて形式検証は行わないことが多い。FPGA は簡単に修正可能であるから、過剰なテストは必要ないという判断だと思われる。FPGA を用いる設計現場がすべてそのようなポリシーという訳ではない。例えば空軍の戦闘機に組み込まれる FPGA を設計している場合、FPGA を書き換える度に大がかりな検査を行わなければならないという事情があり、事前のテストもしっかり行う。

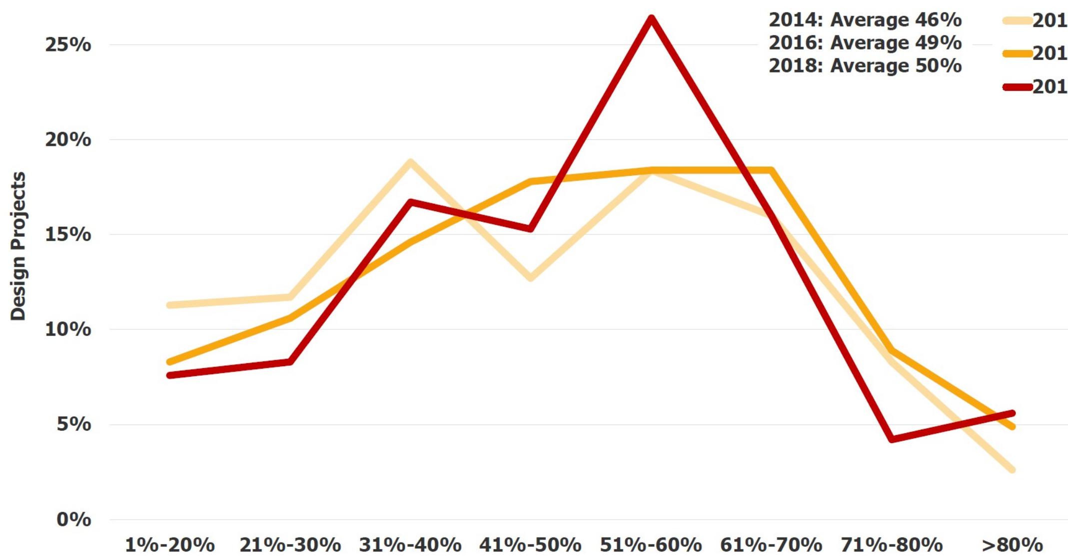
1.2.2 デバッグ・ECO

設計データを変更するシチュエーションとしては、大別してデバッグと ECO の 2 種類が存在する。デバッグとは、設計データ中の誤りを修正することである。前節で説明したテストや検証によってバグを見つけることが出来るので、得られた入力パターンなどから修正すべき部分を特定して修正する。

もう 1 つのシチュエーションは ECO(Engineering Change Order, 設計変更指令) である。これは、仕様そのものが変更されることを指す。市場状況の変化や顧客要求の変化などによって、仕様そのものが変更されることは良くあることである。特にゲートレベル記述などが得られている最終段階で仕様変更が発生し、それに応じてゲートレベル記述を修正し



(a) ASIC/IC development [5]



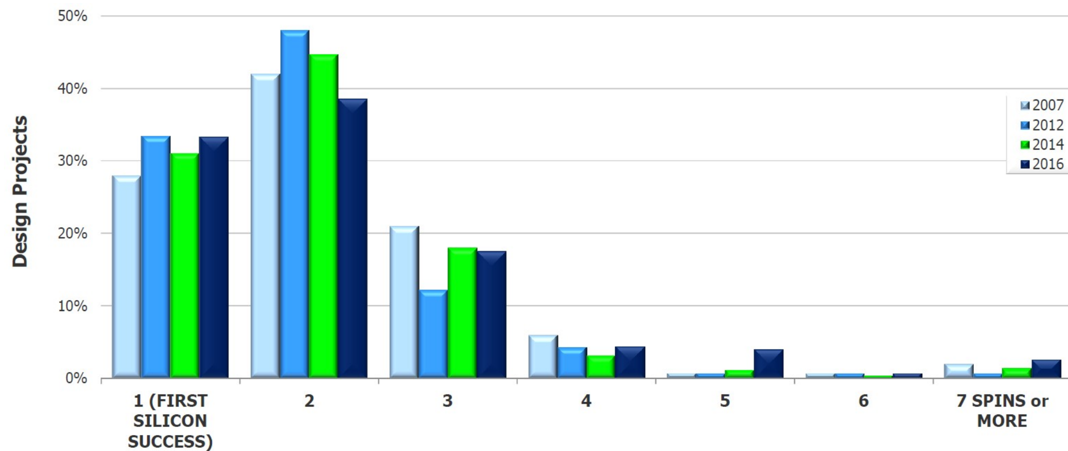
(b) FPGA development [1]

図 1.2: Verification time ratio

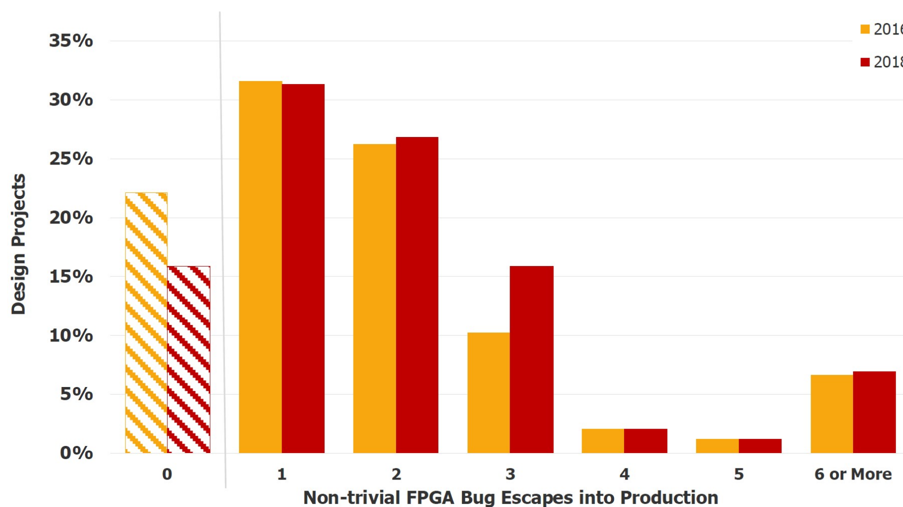
なくてはいけない場合を ECO と呼ぶことが多く、高位設計中に仕様が変更された場合には ECO とは呼ばない。なお ECO は LSI 設計特有のものではなく、工業機械の部品変更など様々な分野で使うことの出来る単語である。

1.2.3　今日の設計現場

タイトルの通り、本論文は回路の論理的な誤りを自動修正する方法について説明する。本節ではこのような手法がいかに重要かを、今日の LSI 設計現場の現状と照らし合わせ



(a) ASIC/IC development [5]



(b) FPGA development [1]

図 1.3: The number of bugs

て説明する。無償で参照できる調査結果を探した結果、本節では [1, 5] を用いることとする。[5] は ASIC/IC 設計現場、[1] は FPGA 設計現場をターゲットにしたものである。それぞれ 1,738 社・1,205 社にアンケート調査を行った結果をまとめている。

図 1.2(a), 図 1.2(b) はそれぞれ ASIC,FPGA プロジェクトでの、全体における検証時間の割合である。ここでの検証とは、設計以外のあらゆる作業（テストやデバッグも含む）を指している。[1, 5] の筆者らは過去に何度か同様の調査を行っており、年ごとの変化が分かる。これによると、どちらの場合も年々検証に用いる時間が増えており、最近ではおよそ 50% 超が費やされている。製品を出荷するまでに、デザインではなく検証に多くの時間が費やされているというのは直感に反するかも知れないが、事実である。

図 1.2(a), 図 1.2(b) だけでは、バグの存在しないプロジェクトでも形式検証ツールの動

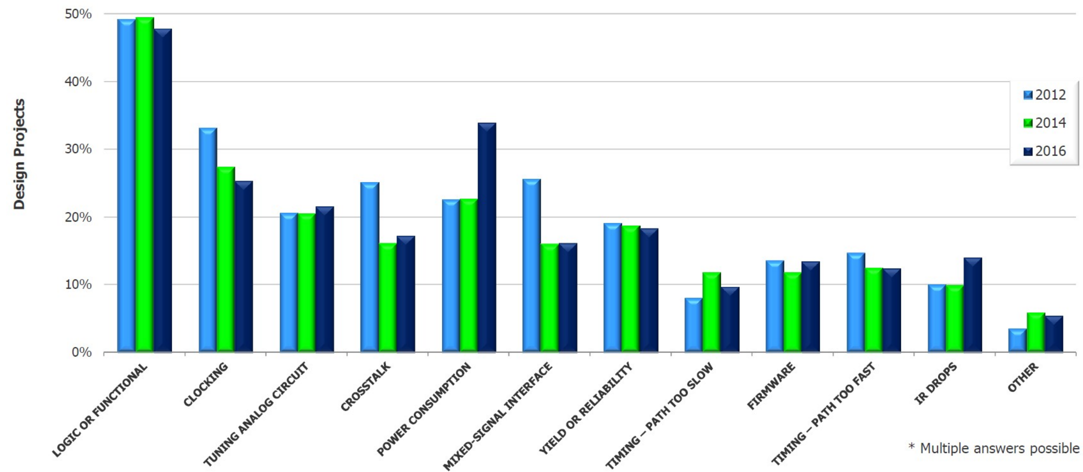
作が遅いせいで時間が掛かり過ぎているだけかもしれない、と思われるかも知れない。これに関連する調査結果として図 1.3(a), 図 1.3(b) を示す。図 1.3(a) は、ASIC 設計で何回マスクが作成されたか（リスピン）を示している。マスクは作成する度に大変なコストがかかるから、バグをすべて取り除いてからにする。それでも多くのプロジェクトでは複数回作成されているのは、マスク発生後にバグが発見されたからである。図 1.3(b) は FPGA に関するデータだが、製品の製造後にいくつのバグが発見されたかを示している。なお、FPGA は容易に設計を変更できるからリスピンの概念はない。ここでも、多くのプロジェクトではバグのない FPGA 設計を作るのは難しいことが分かる。以上のように、多くの設計現場では確実にバグが発生しており、検証時間が単に形式検証ツールの実行時間に占められている訳ではない。

それでは、どのようなバグが発生しているのだろうか。回路設計では、目標とするタイミング制約を満たすための涙ぐましい努力が語られる事が多く、より早い回路を生成するための手法がたくさん提案されている。そのようなタイミング制約絡みのバグが多いのだろうか？その答えを図 1.4(a), 図 1.4(b) に示す。アンケートでは複数のバグを選択することができ、合計が 100% を超えている点には注意したい。順に、ASIC プロジェクト、FPGA プロジェクトでのバグの原因を示している。どちらでも共通して、論理バグが最も多いことが分かる。FPGA プロジェクトでの論理バグは減少傾向にあるが、これは設計現場で形式検証ツールなどが真剣に導入され始めたことが原因であるとのことである。2018 年のデータはおよそ ASIC プロジェクトと同等程度の比率となっており、今後更に論理バグの占める割合が削減されるかどうかは分からない。ASIC プロジェクトでは近年、消費電力に関するバグが増えているようである。調査結果に原因は述べられていないが、直近のプロセスが特に不安定であることや、専用チップ・FPGA を搭載した低消費電力の組込みシステムが増えていていることが原因であると考えられる。

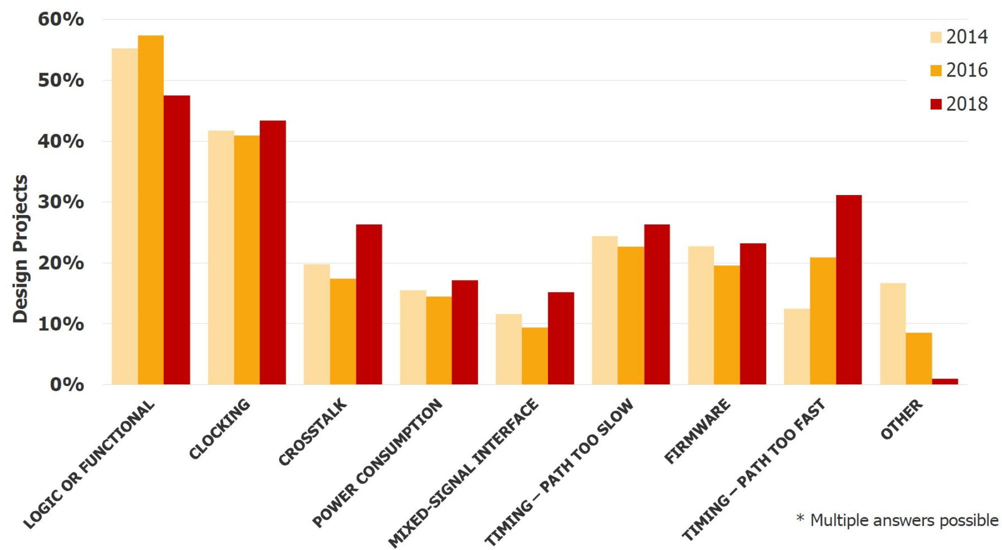
ASIC プロジェクトでは追加のデータがある。図 1.5 は、論理バグの根本原因の内訳が示されている。いくつかの原因を重複して選択できるので、合計が 100% にはなっていない点に注意したい。データからは、単なる人が作った設計データの誤りが最も多いが、仕様の変更や仕様の誤りが原因となった論理の変更 (ECO) も少ないことが分かる。

さて、これまでは検証に掛かる時間やその種類について述べてきたが、実際にどのように時間が使われているかも見てみよう。図 1.6(a), 図 1.6(b) はそれぞれ、ASIC/FPGA プロジェクトで検証を担当するエンジニアがどのような時間の使い方をしたかを示した円グラフである。どちらの場合も、検証でエンジニアにとって最も負担となっているのはデバッグ作業であることが分かる。およそ 40% の時間が実際のデバッグ作業に使われており、それ以外の時間がテストベンチ作成やシミュレーションなどに費やされていた。

以上、今日の設計現場の現状を俯瞰した。まとめると、設計現場ではデザイン以上に検証に時間が取られており、その半数近くは論理バグが原因であることが分かった。検証を



(a) in ASIC/IC development [5]



(b) FPGA development [1]

図 1.4: Reason of bugs

担当するエンジニアは、バグを見つけることではなくバグを修正することに多くの時間が取られる傾向にあることも分かった。従って、論理バグを自動修正する手法は、今日のLSI 設計現場にとって有用であると考えられる。タイトルにあるとおり、本論文では論理的なバグを自動修正する手法を提案する。

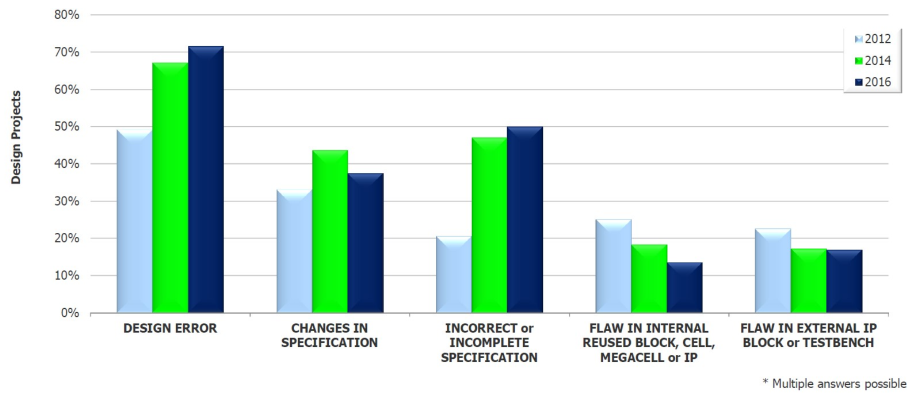
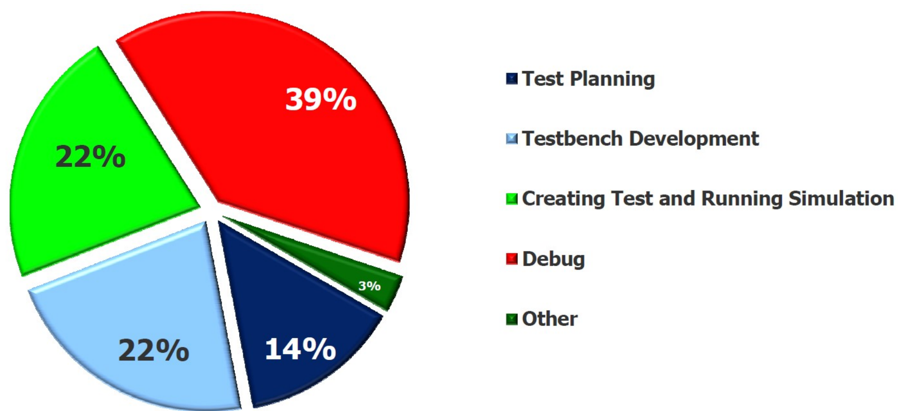
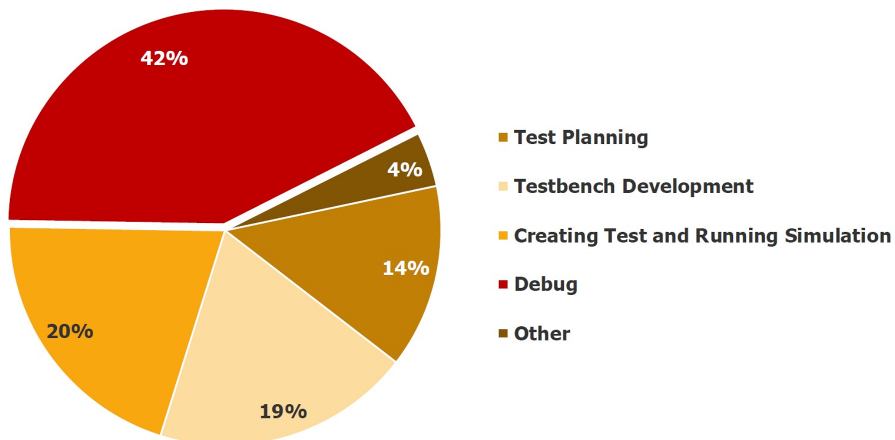


図 1.5: Root cause of logic/functional bugs in ASIC development [1]



(a) ASIC/IC development [5]



(b) FPGA development [1]

図 1.6: Where verification engineers spend their time

1.3　目的

前節で説明したとおり、デバッグや ECO は開発現場では重大な問題であった。本研究では、特に負担の大きな「デバッグや ECO のための自動修正手法」を提案することを目的とする。また、提案手法がどの程度の規模の設計にまで適用可能であるかを示すことも目的とする。

特に困難なデバッグ・ECO の状況としては、以下の２つを想定した。

- （状況１）バグや ECO をもとに、なるべく少ない変更でゲートレベル記述を自動で修正したい場合
- （状況２）C ベース設計において、ゲートレベル記述まで進んでから手動で修正し、それをもとに C 記述を自動で修正したい場合

修正を自動化するには、修正すべき設計記述だけでなく、「仕様」も必要である。人間が修正する場合には、おかしな挙動を示す入力パターンや、同僚からの自然言語による説明で事足りる。しかしこれらは自動化には不十分である。本研究では以下の２つの仕様の与え方を考える。

1. 論理式として与える（状況 1 で利用）
2. シュミレーションにおけるいくつかの入出力パターンが得られる（状況 2 で利用）

ゲートレベル記述を自動でデバッグする手法では、仕様は論理式で与える。詳細は第 1.3.1 節で述べる。C 記述を自動で修正する手法では、仕様はシミュレータとして与える。シミュレータとは実際には、論理シミュレータにゲートレベル記述が与えられたものである。詳細は第 1.3.2 節で述べる。

1.3.1　ゲートレベル記述の自動修正：修正信号選択手法

これは、今日の LSI 設計者が実際に時間をかけて取り組んでいる問題である。C 記述や RTL 記述は人手で作成されることが多いから、そこに混入したバグは人手で取り除けるはずである。しかし、ゲートレベル記述は自動生成されることが多く最適化まで施されてしまうので、信号名が分かりにくくなったりして人手で修正することが難しい。C ベース設計では状態割り当てが自動で行われてしまい、一層分かりにくい。新しい C・RTL 記述を再合成すれば新しいゲートレベル記述を得ることは出来るが、C・RTL 上では小さな変更でもゲートレベル記述上では大きな変更になってしまうことがある。すでに満たしたタイミング制約や回路規模は変更したくないから、そのまま新たなゲートレベル記述を使えることは少ない。

そこで提案手法では、新ゲートレベル記述は「論理式として与えられる仕様」として使うことにし、古いゲートレベル記述を直接自動修正することを考える。この手法にはたくさんの既存研究があるが、以下の 3 ステップに分けて修正が進むことが多い。

1. 修正すべきゲートを発見
2. どの信号（内部信号を含む）を使って修正可能か探索
3. 実際に修正のための回路を生成

本研究では、特に近年盛んに研究されている 2 ステップ目に着目し、修正に十分な信号組を探したりする手法を提案する。組合せ回路・順序回路に分けて手法を提案する。

組み合わせ回路を取り扱った既存手法はいくつか存在するが、本研究では複数の修正すべき場所がある場合でも利用可能な手法を提案する。順序回路を対象とした既存研究もあるが、複雑な前提条件が必要であった。本論文の提案手法は、様々な順序回路に適用可能なものである。

1.3.2　修正されたゲートレベル回路からの C 記述再合成

2 つ目は特殊であり、今日の LSI 設計では一般的でない。しかし、筆者は自動修正手法の新たな応用法と考えている。なお、ここでは C ベース設計を前提とする。図 1.7 も参照されたい。

ゲートレベル記述を設計したあとに ECO が発生した場合、ゲートレベル記述だけでなく C 記述も修正したい。ゲートレベル記述だけ修正してもチップを作成することはできるが、対応する C 記述はその後の検証やシミュレーションにも活用可能であるから、多くの開発現場では両方修正している。しかし C 記述とゲートレベル記述の開発者が異なる場合、ゲートレベル記述を修正出来ても C 記述を修正出来ない場合が考えられる。また、修正箇所がビット演算などの場合には修正そのものが難しくなる。

提案手法では、修正後のゲートレベル記述を「シミュレータとして与えられる新仕様」として使うことにし、古い C 記述を自動修正することとした。なお、ゲートレベル記述は組み合わせ回路か、固定サイクルで出力が得られる順序回路とする。

1.4　本論文の構成

まず 1 章で、本論文の目的などについて説明した。

2 章ではバックグラウンドとなる知識や関連研究について説明する。提案手法は形式手法を活用することになるから、定式化でよく見られる SAT・QBF 問題などについて説明する。さらに、論理回路自動修正手法やプログラム自動合成手法の既存研究を紹介する。

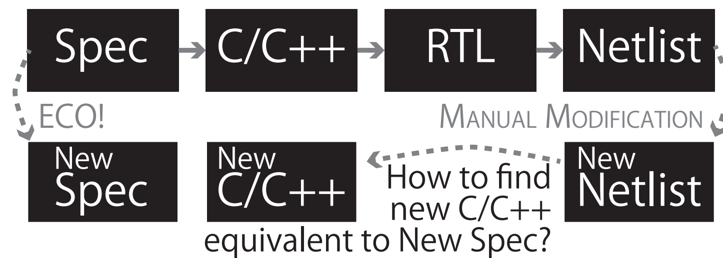


図 1.7: Automatic C modification based on a gatelevel circuit

3 章では、ゲートレベル記述を修正するための修正信号選択について、組合せ回路に限定した手法を説明する。手法では、回路中のどのゲートに修正が必要かはわかっているものとして、どの内部信号を使って新しいパッチ回路を作成するかを考えるものとする。手法の目標は、なるべく少ない数の内部信号組で修正したりすることである。本手法は、修正すべきゲートが複数ある場合でも、品質の良い内部信号組が得られる点が新しい。同様の問題設定のプログラミングコンテストで出題された回路を用いて、既存研究との比較を行う。

4 章では、前章の内容を順序回路にも適用するための手法について説明する。順序回路には到達可能状態と到達不可能状態が存在するが、到達不可能状態まで考えて回路を仕様回路と正しくしようとすると解が得られない場合が多い。提案手法では、到達可能状態組をどのように生成するかが説明されており、正しい信号組を得ることが出来る。ITC99 などのベンチマーク回路を用いて、適用可能な回路規模について考察する。

5 章では、修正されたゲートレベル記述から C 記述を自動修正する手法について説明する。本手法は、デジタル回路を設計した人と C 記述を作成した人が異なる場合や、C 記述で修正すべき部分がビット演算だったりして難しい場合などに有効である。プログラム自動合成手法が LSI 設計で活用可能であることを示す。実験では AES256 暗号化プログラムなどを扱い、適用可能なプログラムの規模について考察する。

最後に 6 章で、結論と今後の課題について述べる。

第 2 章

背景技術

2.1　SAT, QBF, ILP

2.1.1　形式手法と SAT 問題

ソフトウェアやハードウェアの解析は、静的（スタティック）なものと動的（ダイナミック）なものに大別できる。動的解析で最も一般的なのは、様々なテストパターンを用意して出力の正しさを確認するものである。静的解析には様々なものがあるが、ソフトウェア開発で良く用いられるのは、コーディングルール適合不適合の検査である。論理的な正しさを確認している訳ではないが、バグ抑制やコードの読みやすさ維持のために多くの現場で定められている。

別の静的解析手法として、形式検証が挙げられる。本稿では形式手法が多用されるから、そのバックグラウンドを詳細に説明したい。形式手法は、数学的にプロパティが満たされることを確認する方法である。「ある信号と別の信号が同時に 1 にならない」などがプロパティであり、形式手法ではすべての場合でそのプロパティが満たされることを数学的に証明しようとする。もしプロパティが満たされない場合があれば、それを再現するような入力列を提示しその後のデバッグに役立てることができる。

形式手法の中で最も直感的で広く用いられているものに「等価性検証」がある。これは特にハードウェアの論理設計では広く用いられており、２つの設計記述が等価であることを証明するのに使う。入出力がそれぞれ in, out で、２つの設計の入出力関係がそれぞれ $out = F(in), out = G(in)$ で表されるとすると、等価性検証では以下を証明すれば良い。

$$\forall in. F(in) = G(in) \tag{2.1}$$

上式はすべての入力 in に対して $F(in) = G(in)$ であることを確認しなくてはいけないから、いかにも大変そうである。以下の式は、否定をとったものである。

$$\exists in. F(in) \neq G(in) \tag{2.2}$$

否定をとることによって、$\forall$ を消すことができ、適切な in の値を探すだけで良くなった。等価性検証では、２つめの式が満たされない場合に「等価」と、満たされる場合に「不等価」と考えることが可能である。

2.1.2　SAT 問題

2.1.2.1　SAT ソルバの入出力

前節で見たとおり、形式検証では問題を定式化する必要がある。式 (2.2) のように全称記号 $\forall$ がなく存在記号 $\exists$ だけのものは、登場する各変数に真偽のいずれかを割り当てて全体を正しく出来るかどうかを考えればよい。このような問題は SAT 問題（充足可能性

コード 2.1: "Dimacs File"

```
c This is a comment. // "c" is the comment.
p cnf 3 3 // FileType, #vars, #clauses
1 -2 0 // 0 is the prefix.
3 0
1 -3 0
```

コード 2.2: "Result from SAT Solver"

```
SAT // The result is SAT or UNSAT
1 -2 3 0
```

問題）と呼ばれる。式 (2.1) には ∀ があるため SAT 問題ではないが、簡単な操作で SAT 問題にすることが出来た。

SAT 問題を解く際には、SAT ソルバ [6,7] を用いる。SAT ソルバへの問題の与え方には様々な方法が考えられるが、通常 Dimacs 形式のファイルを入力とする。Dimacs 形式のファイルには SAT 問題が CNF(Conjunctive Normal Form、連言標準形) で記述されている。CNF はリテラルを 1 つ以上含む論理和をすべて論理積したものである。リテラルには正リテラル (例えば A) と負リテラル (例えば $\neg A$) の 2 種がある。CNF 中で各論理和部分は「節, Clause」と呼ばれる。例えば以下の式 (2.3) は CNF である。

$$(A \lor \neg B) \land C \land (A \lor \neg C) \tag{2.3}$$

上式を SAT ソルバで解く際には、Dimacs 形式にする必要がある。Dimacs 形式では、各変数に 1 から順に数字を割り当てて問題を記述する。正リテラルはそのままの数、負リテラルは負の数にして節を作る。"c"から始まる行はコメント、"p"から始まる行には順にファイル形式 (cnf)、変数の数、節の数を与える。各節の終わりには 0 を置く。式 (2.3) を DIMACS 形式にしたものをコード 2.1 に示す。A,B,C はそれぞれ 1,2,3 であり、負リテラルにはマイナス符号が付けられる。$A \lor \neg B$ は"1 -2 0"であり、$A \lor \neg C$ は “1 -3 0” である。

SAT ソルバは与えられたファイルが充足可能かどうかを調べる。充足できない場合 UNSAT、充足できる場合には SAT とその割り当てを返す。コード 2.1 を与えた場合の結果をコード 2.2 に示した。$(A, B, C) = (True, False, True)$ とした場合に充足可能であることがわかる。

2.1.2.2　DPLL アルゴリズム [8, 9]

SAT ソルバの基本的な仕組みは、変数に真/偽を割り当てることを繰り返して SAT となるかどうかを探索するというものである。2 分決定木を作って様々に変数を割り当てて、矛盾を発見した時点でバックトラックを行うという処理を繰り返せば、コンピュータで問題を解くことが出来るようになる。式 (2.3): $(A \vee \neg B) \wedge C \wedge (A \vee \neg C)$ を考えてみよう。

1. A=0 を割り当て
2. B=0 を割り当て
3. C=0 を割り当て→矛盾が発生 (C 節)・直前の決定 (C) までバックトラック
4. C=1 を割り当て→矛盾が発生 ($A \vee \neg C$ 節)・変更可能な直近の決定 (B) までバックトラック
5. B=1 を割り当て→矛盾が発生 ($A \vee \neg B$ 節)・変更可能な直近の決定 (A) までバックトラック
6. A=1 を割り当て
7. B=0 を割り当て
8. C=0 を割り当て→矛盾が発生 (C 節)・直前の決定 (C) までバックトラック
9. C=1 を割り当て→解

上述の手順でコンピュータプログラムによる探索が可能であることは示されたが、効率が悪い。例えば、C=1 は自明であるが、0 から順に割り当てるアルゴリズムが採用されているためにバックトラックが必要だった。SAT ソルバで良く用いられる DPLL アルゴリズムは、こういった無駄を極力回避するための効率的な手順やルールを提案している。以下に代表的なルールを示す。

- 単一リテラルの節は値の割り当てが行える
- すべて正リテラル/負リテラルのいずれかしか登場しない変数を含む節は、対象リテラルに値を割り当てた上ですべて削除できる

DPLL を適用することで、式 (2.3) はより効率的に探索できる。

- 単一リテラル節により、C=1 を割り当て
- 新しい簡略化された命題 $(A \vee \neg B) \wedge A$ が得られる
- 単一リテラル節により、A=1 を割り当て
- 命題は常に真となる。B は任意に割り当て。

バックトラックの際には、直近の変更可能な決定を変更するのが直感的である。しかし、現在探索している空間が実際の解を含まない場合、正しい解を得るまでにたくさんのバックトラックを繰り返さなくてはならない。そのようなことがないよう、今日の SAT

コード 2.3: "An additional constraint"

```
-1 2 -3 0 // 1st Addtional Clause
// Solution:
// SAT
// 1 2 3 0
-1 -2 -3 0 // 2nd Additional Clause
// Solution:
// UNSAT
```

ソルバにはリスタート機能が実装されており、適当な間隔で試行する割り当てを大きく変化させるようになっている。

2.1.2.3 CDCL アルゴリズム [10]

DPLL アルゴリズムでは、「何が原因でバックトラックしたのか」ということを解析しないので、命題によっては何度も同じ節の矛盾によってバックトラックを繰り返す必要がある。CDCL アルゴリズムは矛盾から学習を行うアルゴリズムである。

CDCL アルゴリズムでは、Implication Graph を作成して、矛盾が発生してしまうような変数割り当てを学習することが出来る。例えば $B \land \neg C$ が成立すると必ず矛盾してしまう場合には、$\neg B \lor C$ が学習される。この学習節を用いることで、不必要なバックトラックをせずに済む。

問題の規模が大きいと、学習節の数が膨大になってしまうことがある。今日のソルバでは、不要と思われる学習節は条件に従って削除される。

2.1.2.4 Incremental SAT

SAT ソルバは充足可能な割り当てのうちのどれか 1 つを返す。もしすべての割り当てを網羅したい時には、得られた解を除外するような節を追加すれば良い。例では $(A, B, C) = (True, False, True)$ が解の 1 つとして得られたから、"-1 2 -3 0"という節を追加すれば良い。さらに得られた解を除外することを繰り返すと、いつか答えが UNSAT となってすべてを網羅することが可能である。以上のような流れをコード 2.3 にも示した。

節を追加する際には、初めから問題を解き直すのではなく、これまでに得られた割り当てや学習節を再活用するのが望ましい。多くのソルバには Incremental モードが備わっており、あとから節を追加することが可能である。Incremental モードを使用すれば、ファイルをはじめから読み直さない上に、これまでの計算結果を再利用するので、高速に問題を解くことができる。

2.1.3　2QBF 問題

2.1.3.1　概要

QBF 問題とは、∀ を含む命題を指す。特に 2QBF 問題は $\forall x \exists y.\phi$ の形で表されるものである。形式検証は 2QBF で表現されることが多く、この問題をいかに効率的に解くかが重要である。なお、以下の 2 つの命題は等価であるから注意したい。

$$\forall x \exists y.\phi \Leftrightarrow \neg \exists x \forall y.\neg \phi \tag{2.4}$$

QBF 問題は、QBF ソルバで解くことが出来る。しかし、2QBF 問題については、SAT ソルバを繰り返し用いることで同等かそれ以上の性能を出すことが出来ることがあり [11]、その方法を次節で説明する。

2.1.3.2　SAT ソルバを用いた解法

$\exists x \forall y.F(x, y)$ という 2QBF 問題を SAT ソルバで解く方法を考えたい。命題は ∀ を含むから、そのままでは SAT ソルバでは解けない。最も簡単な方法は、すべての y について式を書き下してしまうことである。y が $y_0 \ldots y_n$ という値をとるならば、以下の式 (2.5) のようにして SAT 問題にできる。式 (2.5) は命題と等価であるから、繰り返し実行したりする必要はない。

$$\exists x.F(x, y_0) \wedge F(x, y_1) \wedge \ldots \wedge F(x, y_n) \tag{2.5}$$

式 (2.5) は ∀ の部分の変数の値の数だけ式を展開しなくてはいけないので、規模が大きくなりすぎるという問題点がある。まずはじめに以下の式を解き、徐々に反例を積み重ねていく解き方も存在する。

$$\exists x, y.F(x, y) \tag{2.6}$$

上の命題を満たすような $(x, y) = (x_1, y_{tmp})$ は、QBF 問題をすべて展開するよりは問題規模が小さくなって見つけやすいはずである。そこでまずは、x_1 が命題を満たすかどうか確認する。これは、以下の式 (2.7) が UNSAT になることで確認できる。

$$\exists y.\neg F(x = x_1, y) \tag{2.7}$$

上式が SAT となった場合、x_1 は解として相応しくない。x に相応しい別の値を探索するために、条件を追加した以下の問題を解く。y_1 は、先ほど得られた y_{tmp} とする。

$$\exists x, y.F(x, y) \wedge F(x, y_1) \tag{2.8}$$

以上のようにして y_i を積み重ねていくことで、次のような命題を繰り返し解くことになる。

$$\exists x, y.F(x, y) \wedge F(x, y_1) \wedge \ldots \wedge F(x, y_m) \tag{2.9}$$

表 2.1: Example of Set Covering

	A	B	C	D	E	F	G	H	I	J
1	*		*	*			*			*
2	*			*	*				*	*
3			*							
4						*				*
5			*			*			*	

y_i の数が増えることによって、徐々に式 (2.9) は式 (2.5) の形に近づいてゆき、最悪の場合等価になる。一般に、すべての y を網羅する前に命題は UNSAT になることが期待できるため、解くべき問題の規模が小さくなって解の導出が高速化することが期待できる。一方で、元の問題の大きさがあまり大きくなく、y の取り得る値の数も小さな場合、あえて何度も小さな SAT 問題を解くよりは、式 (2.5) を解いた方が高速である。

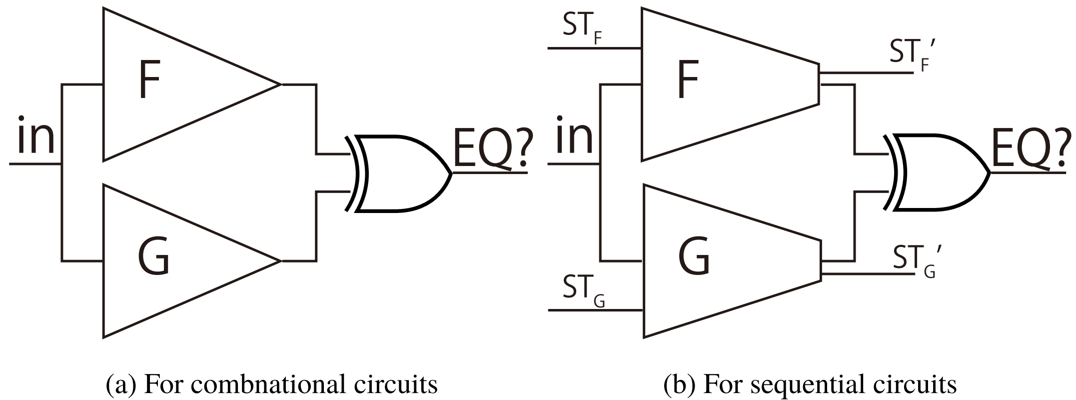
2.1.4 ILP 問題

LP 問題（Linear Programming, 線形計画問題）は、制約条件や目的関数がすべて線形に表される問題で、特に ILP 問題（Intenger Linear Programming）は解が整数に限定されたものを言う。LP は最悪でも多項式時間で解くことができるが、ILP には当てはまらない。本研究では信号選択手法を ILP 問題（セットカバリング）として定式化するから、その例と簡易な解法を述べておく。

ILP 問題として有名なものに、セットカバリング問題やナップザック問題などがある。セットカバリング問題では、すべての行をカバーするように、なるべく少ない列を選択することが目的である。

表 2.1 には、各行がどの列によってカバーされるか、"*"で示されている。行 3 は列 C にだけカバーされるから、列 C は必ず選択しなければならない、すると自動的に行 1 もカバーされるから、行 1,3,5 だけを考えれば良い。残りの行をカバーするのは、(I,J),(A,F) など様々な組み合わせが挙げられる。今回の例で行 1,3,5 を 1 つの列でカバーするのは不可能だから、すべてをカバー可能な列の数は C を含めて 3 である。以上のように、カバーする列を挙げるのは簡単だが、列数が最小の組み合わせを見つけるのは難しく、NP 困難である。従って、いくつかの近似解法が存在する。

1 つめは「貪欲法」である。貪欲法では、なるべく多くの行をカバーするような列を順番に選ぶことによって解を得る。最適解であるかどうかはわからないが、答えが大きすぎ



(a) For combnational circuits (b) For sequential circuits

図 2.1: Miter

ない妥当な解が出ることが多い。もう 1 つは、問題を LP 問題として解き、その結果を参考にすることである。これは線形緩和と呼ばれる。LP 問題は多項式時間で解けるから、それを元にすることで最適解に近い解が早く求まると考えられる。商用の ILP ソルバには、最適解と、時間内に得られた最良の許容解を得る方法がそれぞれ実装されている。商用ソルバは一般に品質の良い解を短時間で出すことが知られており、本研究でも貪欲法と商用 ILP ソルバを組み合わせて解を高速に得られるようにした。

2.2 論理回路の等価性検証

2.2.1 概要

前節でも紹介したが、等価性検証とは 2 つの回路設計記述が等価であることを証明することである。2 つの回路が組み合わせ回路であると仮定し、入出力がそれぞれ in, out、入出力関係がそれぞれ $out = F(in), out = G(in)$ で表されるとすると、等価性検証では式 (2.11) を調べれば良い。UNSAT ならば等価、SAT ならば不等価である。

$$\forall in.F(in) = G(in) \tag{2.10}$$

$$\Leftrightarrow \exists in.F(in) \neq G(in) \tag{2.11}$$

SAT ソルバが取り扱える問題の規模はせいぜい数百万変数と言われているので、ゲート数が百万程度の回路ならば上式がそのまま適用可能である。上式を回路図にしたものを図 2.1(a) に示す。このような回路は特にマイターと呼ばれる。

商用の等価性検証ツールは、2 つの回路のモジュールやブロック間の対応情報を用いて、複数の小さな等価性検証を行いながら回路全体の等価性を示すことを行っている。これによって高速化が図れる。

２つの回路が順序回路である場合は難易度が高くなる。順序回路のマイターを図 2.1(b) に示す。組み合わせ回路との違いは、FF 由来の入出力が増えている点である。順序回路にはフリップフロップ（FF）があるが、その数や状態遷移が 2 回路間で同じとは限らない。ST_F, ST_G に任意の値を与えてしまうと、実際にはあり得ない状態組でまで等価性検証をしてしまうことになる。仮に 2 回路の FF と状態遷移が全く同じだったとしても、到達不可能な状態での論理が異なる場合、単に組合せ回路の等価性検証では正しい答えが得られない。従って、順序回路の等価性検証は簡単には行えない。商用の等価性検証ツールはこのような問題を自動で解決することは諦めており、合成時に得られる中間情報（状態の対応や到達不可能状態に関する情報）が必須の入力であることが多い。

高位合成ツールには、高位記述と RTL 記述間の等価性検証ツールが付属することがある。一般に、高位記述は状態割り当てが行われていないから、そのまま RTL 記述と等価性検証するのは難しい。このような場合にも、合成時の中間情報（C 記述内のステートメントと RTL 記述内のブロックの対応など）が活用される。

2.2.2　順序回路の等価性検証

２つの順序回路が等価であるとは、「同じ入力シーケンスに対して同じ出力シーケンスが得られる」ものとする。

2.2.2.1　直感的な方法

図 2.1(a)(b) にあるようなマイターを考える。入力状態はそれぞれ ST_F, ST_G であり、対応する次状態は ST'_F, ST'_G とする。初期状態はそれぞれ既知と仮定する。

直感的に思われる方法は、初期状態から 1 つずつ列挙していくものである。回路の入出力を in, out、入力側の状態組を $pair_{current} = (ST_F, ST_G)$、出力側の状態組を $pair_{next} = (ST'_F, ST'_G)$ とする。次状態は $pair_{next} = Miter(in, pair_{current})$ で得られるとする。ある状態 $pair_{current} = current$ から到達可能な次状態組 $Reachable$ は、以下の SAT 式を繰り返し解くことによって得られる。

$$\begin{aligned}\exists in, current, next \\ next &= Miter(in, current) \\ \wedge current \in Reachable &\wedge next \notin Reachable \end{aligned} \tag{2.12}$$

Reachable は $next$ を追加することで更新できるので、毎回新たな $next$ が得られる。上式が UNSAT になれば、1 サイクル次の状態はすべて網羅したことになる。調べていない Reachable 内の状態をすべて $current$ として繰り返せば、いずれすべての到達可能状態組を得ることが出来る。最後に、得られた到達可能状態組だけでプロパティを検証すれば、順序回路同士の等価性が分かる。

2.2.2.2　PDR [12]

直感的な方法では到達可能状態組の数だけ SAT 式を解かなくてはいけないから、現実的な時間内に終わらない可能性がある。

PDR(Property Driven Reachability) は、到達可能状態組のスーパーセットを用いてプロパティの真偽を評価する手法である。プロパティには様々なものか考えられるが、2 回路の等価性を例として考えよう。状態組のスーパーセットで常に等価な出力を得られるなら、実際の到達可能状態組も常に等価な出力を得られるはずだから、等価である。スーパーセットで等価な出力が得られない時には、スーパーセットが大きすぎる可能性がある。より詳細に反例を解析して小さなスーパーセットを探すか、不透過になる状態組が初期状態組から到達可能であることが示せばよい。

UNSAT 時の必須の節（UNSAT Core）や、SAT 時の必須の変数割当ては、状態組をいくつかまとめて使うのに有効である。到達可能状態組のスーパーセットや到達不可能状態組のサブセットは、これらを活用するのと相性が良く、PDR で活用されている所以である。

既存研究によると、PDR による順序回路の等価性検証では、50 個程度の FF が含まれる順序回路も取り扱うことが出来る。

第 3 章

組合せ回路のための修正信号選択手法

修正した回路を再合成し直すと、以前のゲートレベル回路とは大きく異なる回路が得られてしまうことがある。設計が進んだ製品では、修正したいゲートレベル回路のサイズや遅延情報・消費電力がすでに他の設計で活用されている可能性があり、それらを大きく変更するのは困難である。従って、すでにあるゲートレベル回路をなるべく小さな変更で修正したいというニーズが存在する。本節ではそのような状況を想定し、直したい回路素子が分かっている際にどの内部信号を利用すれば回路が効率的に修正できるか探索する手法について説明する。

3.1　既存研究

ゲートレベル回路に誤りがあることが分かった場合、一般的に次の 3 ステップに沿って修正が行われる。

1. 修正すべきゲートを発見
2. どの信号を使って修正するか探索
3. 実際にパッチ回路を作成

多くの研究では、第 1 ステップの結果（どのゲートを修正すべきか）は所与のものとして扱われている。商用の合成ツールでは、入力ファイル (Verilog ファイルなど) の各ステートメントが、合成後のファイルのどの部分に対応するかが保存されている。従って、どの部分を直せば良いかはツールが予め知っていることが多い。そのような情報がない場合でも、バグを再現するような入出力パターンやランダムパターンを活用することで、直すべき場所が予想できる。

信号の探索はせずに、修正すべきゲートの論理だけを変更して仕様と等価にしようとする研究も存在する [11]。複数のゲートを選んで上手に修正すれば等価に出来ることもあるが、これだけでは不十分な場合もある。従って、本研究では修正に使う信号も選び直す必要があるものとする。

3.1.1　既存研究の動向

初期の研究 [13, 14] では、以下の 3 つが入力として与えられ、出力としてパッチ回路が得られる。

- 正しい RTL
- バグのある RTL
- バグのあるゲートレベル回路

２つの RTL の差異から直すべき場所が推定され、パッチ回路が生成される。これらの研究の欠点は、修正に当たってプライマリー出力のみを用いる点である。内部信号を使用しないので、パッチ回路が大きくなりすぎてしまう問題点がある。より小さなパッチ回路を生成しようとした研究としては [15] が挙げられる。[15] では新設計と旧設計の構造的（幾何的）な違いを用いて旧設計記述を修正が提案されている。

２ステップ目を考慮した研究が近年盛んとなっており、[2,16–20] が挙げられる。

[17] では、修正すべき場所が複数の場合についても実験が行われており、数十万ゲートの組合せ回路で数分以内に結果が得られている。この論文の筆者らは、同様の問題設定の ICCAD’17 CAD プログラミングコンテスト [21] で優勝している。[2] ではパッチを作成するために必要な信号の選択が集合被覆問題として定式化されており、なるべく少ない数の信号を用いてパッチが生成できる。[18] は前述の手法をより最適化しており、発見的な手法を用いて信号選択を効率化している。[16] は SPFD を用いて信号選択を行っている。SPFD は FPGA 設計の最適化を目的として提案された回路の表現方法だが、ここでは信号選択に応用されており、集合被覆問題とは異なったアプローチである。

3.2　目的

本章では、バグのあるゲートレベル記述を自動修正することを考える。前節の既存研究紹介で、ステップ 2 の研究が近年盛んに行われていると述べた。本章では、複数の直すべき場所（ターゲット）がある回路でも使用可能な修正のための内部信号選択手法を提案する。貢献点は以下の２つである。

- 複数のターゲットを同時に修正するための定式化を提案した。提案手法が ICCAD’17 コンテストベンチマーク回路において高品質な解を得られることを確認した。
- 複数同時に直せない場合には、１つずつ直す既存手法が必要である。実験から直す順番が重要であることがわかり、回路の幾何学的な構成から順番を決める手法を提案した。同様のベンチマーク回路を用いて実効性を確認した。

なお、直すべきゲートレベル記述は組み合わせ回路であるものとし、順序回路のための手法は第 4 章で別に紹介する。順序回路の組み合わせ部分だけを修正したい場合や、順序回路で 1 サイクルで演算が終わる場合にも、本章の提案手法が適用可能である。

3.3　問題の定義

次の 3 つ（オプションとしてもう 1 つ）が入力として与えられるものとする。

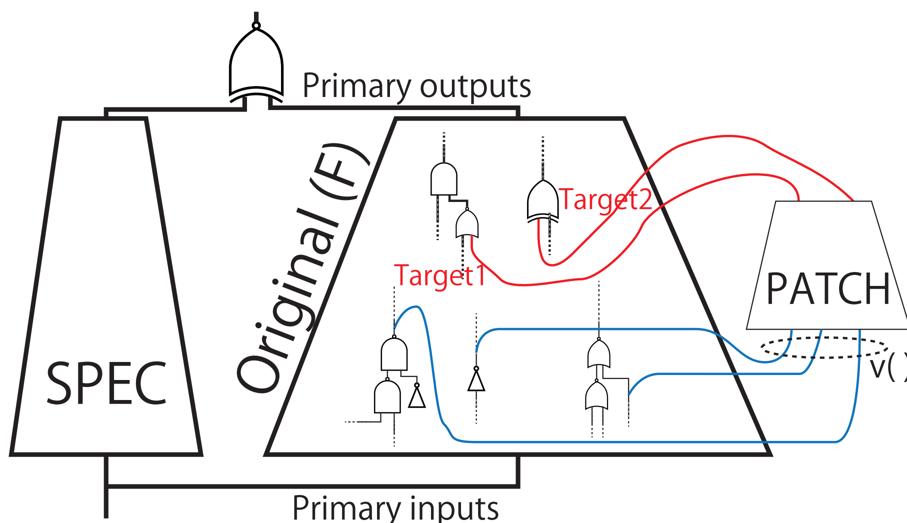


図 3.1: Problem Definition

- 満たすべき仕様（ゲートレベル記述として与えられる）
- 修正したいゲートレベル記述
- 修正したいゲートレベル記述内で、 1 つまたは複数の直すべきゲートの場所（ターゲット）
- （任意）修正したいゲートレベル記述内の、各信号の重み

前述の通り、回路はすべて組み合わせ回路であると仮定する。仕様回路と修正したいゲートレベル記述の入出力の対応は分かっているものとする。仕様はゲートレベル記述として与えられるので、読者はもう回路を修正する必要はないと思うかも知れない。前述の通り、仕様回路が修正したい回路とは大きく異なるトポロジーの場合、そのままでは使えない。従って、修正したいゲートレベル記述の構造をなるべく保ったまま、論理が仕様回路と等価になるように直す必要がある。修正すべきゲートの場所は「ターゲット」と呼ばれ、 1 つでも複数でも良い。ターゲットは誤った論理を持っているから、正しい回路を接続し直す必要がある。このような回路をパッチ回路と呼ぶ。ゲートの種類を変えるだけでは十分でなく、ゲート入力の変更も必要である点に注意したい。本章では、ターゲットの場所は正しいものと仮定しており、どんなパッチ回路でも直せない場合は考えない。各信号の重みを与えることができるが、ない場合にはすべて重みが 1 であるものとして考える。図 3.1 も参照されたい。

本章で提案する手法の目的は、「パッチ回路のための最適な入力組を探す事」である。最も簡単な直し方としては、パッチ回路にすべての回路入力を接続することである。回路は入力の関数だから、必ず正しいパッチ回路が得られるはずである。しかしこれでは必要以上に大きなパッチ回路が出来る可能性があるから、なるべく内部信号を活用したい。そ

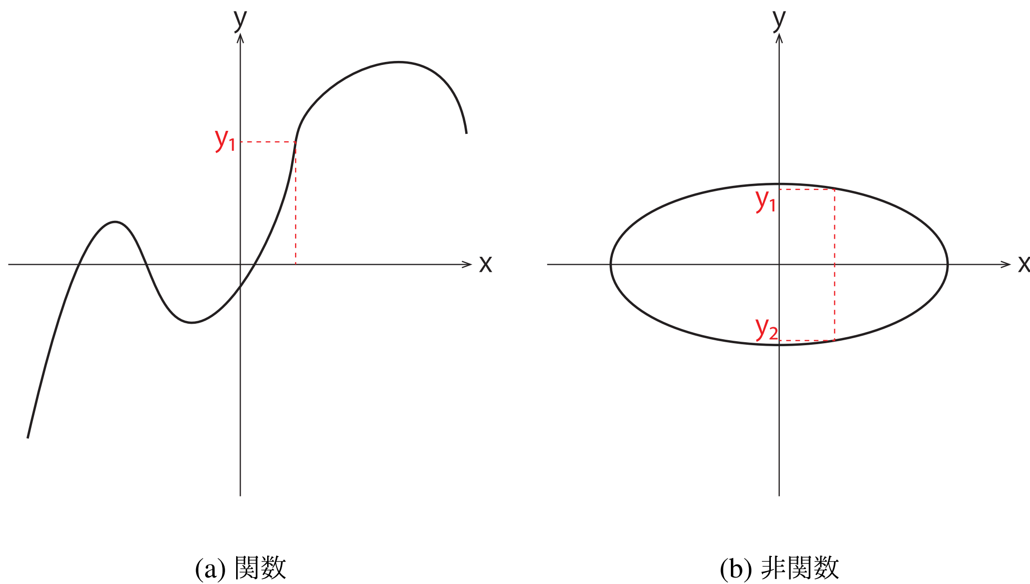


図 3.2: Function(a) and Non-Function(b)

こで、なるべく合計重みが小さくなるような入力組が最適であるものとする。すべての重みが 1 の場合、信号数を最小にすることになる。重みがターゲットからの距離に依る場合、配線しやすい信号組を探す事になる。本稿ではパッチ回路の大きさは考慮しないから注意されたい。

3.4　複数ターゲットを同時に扱う定式化

3.4.1　アイデア

数学における関数とは、入力に対して出力が 1 つに定まる対応のことを言う。入力 x、出力 y を持つような関数をグラフに表した時に、図 3.2(a) のように入力に対してただ 1 つの出力が定まれば関数である。一方で図 3.2(b) では 1 つの入力に対して 2 つの出力が考えられ、関数ではない。

論理関数も関数であり、論理回路を生成する際には仕様が明らかでなければならない。同じ入力値に対して異なる出力値を取る必要があるような回路は不適切であり、回路として実現することが出来ない。

提案手法は回路を修正するための内部信号を自動選択するものだが、選択にあたって「関数の定義」を活用し、すべての入力値が適切な 1 つの出力値を得られるように定式化するものとする。

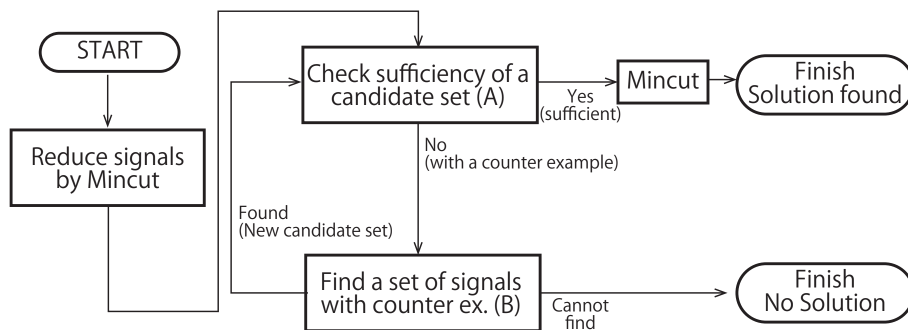


図 3.3: Overall flow of the method for single target problems [2]

3.4.2　単一ターゲットのための既存手法

本節では、提案手法を解説する前に既存手法 [2] を説明する。この手法は提案手法の元となるものである。問題の定義は 3.3 節とほぼ同じだが、ターゲットは 1 カ所のみとする。手法は単一ターゲットでのみ利用可能なものである点に注意されたい。

手法の概要を図 3.3 に示す。提案手法では主に (A),(B) の 2 ステップの繰り返しで構成されている。(A) では今得られている信号組が十分かどうかを確認し、十分でない場合には追加の反例に基づいて (B) で新たな信号組が探索される。候補となる信号数に応じて (B) の処理速度が大きく変化するため、(A),(B) の繰り返し前にグラフの最小カットを用いて探索範囲の枝刈り行われている点に注意されたい。解が得られた後にもグラフの最小カットを用いてより良い解 (合計重みが小さな解) が得られる可能性がある。

3.4.2.1　信号組の十分性の確認 (A)

修正したい回路 F を $out = F(in, t)$ と表現する。in, out がそれぞれ入出力値であり、t はターゲットの値とする。ここでは単一ターゲットであるから、$t = 0, 1$ のいずれかである。仕様となる回路 G は $out = Spec(in)$ と表し、in, out は同様に入出力値である。候補となる信号組は v とし、入力値 in に対応する v の値を $v(in)$ と表現する。信号組は 1 つ以上の信号をまとめたものであり、$v(in_1) = v(in_2)$ は対応する信号の値がすべて等しくなることを指す。$v(in_1) \neq v(in_2)$ は、対応する信号のうちの少なくとも 1 つが異なる値となることを指す。$v()$ にあたる信号組は (B) で得られる。

内部信号には、プライマリ入力 x の影響しか受けないものと、t によって値が変化するものがある。パッチ回路によってループが生じてはいけないから、パッチの入力候補は t の Fanout コーンに含まれてはいけない。従って、パッチ入力候補となるような内部信号の値はすべてプライマリ入力のみの関数で表され、$v(in, t)$ ではなく $v(in)$ となっていることに注意したい。

$v()$ が十分であるとは、それらの信号のみでパッチ回路が作れ、$\forall in.F(in, t = p(v(in))) =$

$SPEC(in)$ を満たす関数 $p()$ が存在することを意味する。

定理 1. 以下の式 *(3.1)* が満たされるとき、$v()$ は修正に不十分である。以下の式 *(3.1)* が満たされないとき、$v()$ は修正するのに十分である。

$$\begin{aligned}
&\exists in_1, in_2 \\
&\quad F(in = in_1, t = 0) = Spec(in_1) \wedge F(in = in_1, t = 1) \neq Spec(in_1) \\
&\wedge F(in = in_2, t = 0) \neq Spec(in_2) \wedge F(in = in_2, t = 1) = Spec(in_2) \\
&\wedge v(in_1) = v(in_2)
\end{aligned} \tag{3.1}$$

証明. 上式で in_1 は $t = 1$ の時だけ直せる入力、in_2 は $t = 0$ の時だけ直せる入力である。ターゲットが異なる出力を持つなら、その入力も異なる値でなければならない。最終行の $v(in_1) = v(in_2)$ はこれに矛盾しており、式 (3.1) が満たされるときは $v()$ は修正に不十分である。また、式 (3.1) が満たされないときにはすべての in_1, in_2 の組合せについて関数の定義を満たしていることになり、$v()$ は十分である。 □

式 (3.1) は ∀ を含まない SAT 式であるから、SAT ソルバを用いて解くことが出来る。

3.4.2.2　新しい信号組の候補生成 (B)

式 (3.1) が SAT となった場合、反例として in_1, in_2 を得ることが出来る。in_1, in_2 では、ターゲットはそれぞれ異なる値をとらなくてはならないのに、既存の候補 $v()$ は等しい値を取ってしまっている。そこで内部信号の中で in_1, in_2 時に異なる値となっているものを探し、候補信号に追加することを考える。

例として、プライマリ入力 A_0, A_1, A_2・内部信号 B_0, B_1, B_2 を持つ回路を考える。in_1 として $(A_0, A_1, A_2) = (0, 0, 1)$、$in_2$ として $(A_0, A_1, A_2) = (0, 1, 1)$ が得られているとする。内部信号の値も含めて表 1 に示す。

表 3.1: Finding an input for the patch

	A_0	A_1	A_2	B_0	B_1	B_2
IN_1	0	0	1	1	1	0
IN_2	0	1	1	0	1	1
$IN_1 \oplus IN_2$	0	1	0	1	0	1

最終行に信号の値の排他的論理和を取ったものを記した。この行が 1 になっている信号 A_1, B_0, B_2 が信号の候補である。このうち 1 つを任意に選んで候補とすれば良い。各信号に重みが割り当てられている場合、最も重みが小さい信号を選ぶのが良い。

さて、このように排他的論理和をとったデータを記録しておき、表 3.2 が得られたとする。すべての行の 1 を少なくとも 1 つカバーするように信号を選ぶと、それが入力の候補となる。3 行目は A_1 によってのみカバーされるので、これは必須である。すると、残る 2 行目をカバーするだけで良いことになり、A_0, B_0 のうちの 1 つを候補とすれば良い。従って $(A_0, A_1), (A_1, B_0)$ のどちらかが入力の候補となる。

表 3.2: Finding inputs for the patch

A_0	A_1	A_2	B_0	B_1	B_2
0	1	0	1	0	1
1	0	0	1	0	0
0	1	0	0	0	0

上述のような問題は集合被覆問題（Set Covering Problem、セットカバリング）と呼ばれている。各信号に重みが設定されている場合、合計重みを最小化するようにする。このような問題は重み付き集合被覆問題（Weighted Set Covering）と呼ばれる。すべての信号の重みが 1 の場合は単なる集合被覆問題となり、信号数を最小化することが目標となる。この問題は ILP ソルバを用いて解くことが一般的である。最適解が得られる必要性はないが、最終的に得られる信号組は ILP ソルバによって最適解が保証されることが望ましい。

なお表 3.2 ですべて値が 0 の行が得られた場合、どんな信号でも回路が修正出来ない。ターゲット信号を操作するだけでは回路が修正できないことになる。

3.4.2.3　最小カットを用いた手法の高速化（提案手法）

前節で信号選択がセットカバリング問題であることを説明した。すべての行をカバーするような信号を選択するだけならば求解には大した時間はかからないが、重みが最小となるような解を求めるには厳密手法・近似手法いずれも時間がかかる。特に信号数（列数）は非常に重要であり、これがこれが 1000 を超えると格段に実行時間が長くなることが実験的に分かっている。従って、不必要な探索空間を予め排除したり、近似手法からより良い解を高速に求める手法が重要である。本節では、グラフの最小カットを用いた手法を 2 つの場面に適用することを考える。

1 つ目は、図 3.3 で繰り返しループが始まる前に不要な信号を除去する手法である。図 3.4 にあるように、回路を有向グラフとして考えよう。各信号の重みがエッジの重みに対応するものとする。ある信号の入力コーン内に合計重みがより小さなカットが存在する場合、その信号を選択するよりも、カットに属する信号を選択した方がよいことになる。入力コーンのカットは、出力信号の論理を必ず表現出来るはずだから、回路として考えても

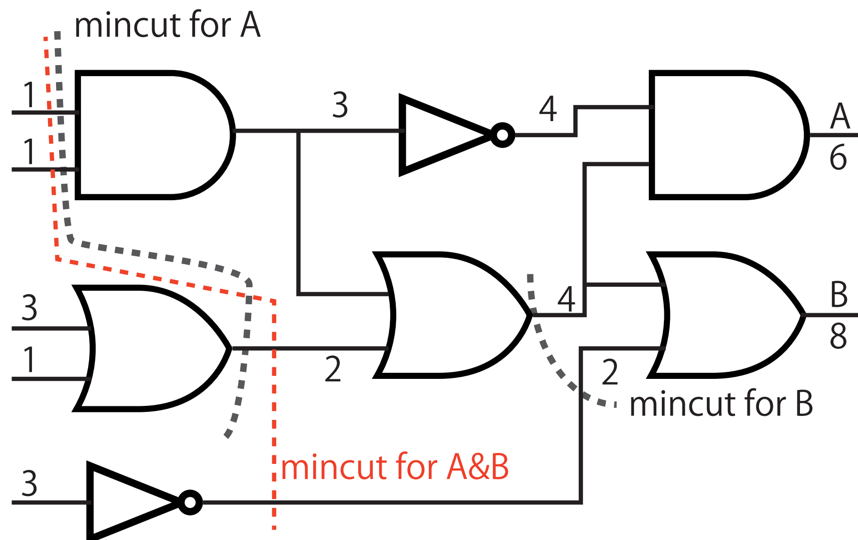


図 3.4: Minimum Cut

問題ない。従って、すべての内部信号について最小カットをもとめ、信号重みよりも最小カットの合計重みが小さければ、その信号はセットカバリング表に必要ないことになる。例えば図 3.4 では信号 A の重みは 6 だが、合計重み 4 となるような最小カットが存在するから、信号 A はセットカバリング表には不要である。

2 つ目は、図 3.3 で繰り返しループが終わって正しい信号組が得られた場合に用いる手法である。セットカバリング問題を解く際にソルバが近似手法を用いている場合、解の合計重みは最適でない可能性がある。信号組の入力コーンをまとめた有向グラフの最小カットをさがし、この合計重みが元の合計重みよりも小さければ、より小さい方を用いれば良い。厳密手法を用いると時間が掛かりすぎてしまう場合でも、最小カットに基づいた信号組である程度最適化することが出来る点で有用である。例えば図 3.4 で A,B が解となる信号組として得られた場合、実際にはより小さな最小カット 6 が存在するので、こちらを解として方がよい。なお、最小カットに基づいた信号組が最適解とは限らないことには注意したい。最小カットではあくまで元の信号組の入力コーンしか考えていないから、入力コーン外に真の最適解が存在する場合は最適解が得られない。

最小カットと最大フローは等しいことが知られており、最小カットの導出には最大フローが用いられる。最大フローを求めるアルゴリズムには様々なものがあるが、例えば計算量 $O(VE\log(V^2/E))$ のアルゴリズムが存在する。ただし、V はノード数、E はエッジ数である。計算量は回路の規模に対し急速に増大してしまうが、実際には上述の計算は大きな回路でも合計で数秒程度で終了し、十分に速かった。

3.4.2.4　パッチの生成

本研究には直接関係しないが、簡単にパッチの生成の仕方も述べる。v がどのような値の時に出力を 1 とすれば良いか考える。出力を $t = 0$ としたときに回路が正しく出来ない場合を網羅すれば良いから、以下の式を繰り返し解けば良い。ただし、すでに得られている出力 1 とすべき $v()$ の値の組（オンセット）は $\vec{On}$ とする。

$$\begin{aligned} &\exists in.F(in, t = 0) \neq Spec(in) \\ &\wedge v(in) \notin \vec{On} \end{aligned} \tag{3.2}$$

すべてのオンセットが得られると、LUT として回路を合成することが出来る。

3.4.3　複数ターゲットのための定式化（提案手法）

本節では提案手法を説明する。

ターゲットが 2 つの時を考えてみよう。ターゲットの値はそれぞれ t_0, t_1 とし、それ以外の記号はすべて式 (3.1) と同様であるものとする。式 (3.3) が UNSAT となれば信号組 $v()$ は十分であり、SAT の時は不十分である。

$$\begin{aligned} &\exists in_1, in_2. \\ &\quad (F(in_1, t_0 = 0, t_1 = 0) \neq Spec(in_1) \vee F(in_2, t_0 = 0, t_1 = 0) \neq Spec(in_2)) \\ &\quad \wedge(F(in_1, t_0 = 0, t_1 = 1) \neq Spec(in_1) \vee F(in_2, t_0 = 0, t_1 = 1) \neq Spec(in_2)) \\ &\quad \wedge(F(in_1, t_0 = 1, t_1 = 0) \neq Spec(in_1) \vee F(in_2, t_0 = 1, t_1 = 0) \neq Spec(in_2)) \\ &\quad \wedge(F(in_1, t_0 = 1, t_1 = 1) \neq Spec(in_1) \vee F(in_2, t_0 = 1, t_1 = 1) \neq Spec(in_2)) \\ &\quad \wedge v(in_1) = v(in_2) \end{aligned} \tag{3.3}$$

上式は特別な in_1, in_2 を探している。前半の 4 行は、同じ t の値で in_1, in_2 を同時に修正出来ないことを条件としている。このような場合には t は異なる値でなくてはならないから、$v(in_1) \neq v(in_2)$ であるべきだが、最終行では矛盾する条件を加えている。従って、上式が UNSAT ならば関数の定義が満たされず、信号組はそのままでは不十分であることになる。このように、$(t_0, t_1) = (0, 0)$ から $(1, 1)$ まですべての組合せについて同時に修正されないような条件を書き下せば良いことが分かったから、２つに限らない複数ターゲットのための条件は以下のように書ける。

定理 2. 以下の式 *(3.4)* が満たされるとき、$v()$ は修正に不十分である。以下の式 *(3.4)* が満たされないとき、$v()$ は修正に十分である。

$$
\begin{aligned}
&\exists in_1, in_2 . \forall t . \\
&\quad (F(in_1, t) \neq Spec(in_1) \vee F(in_2, t) \neq Spec(in_2)) \\
&\quad \wedge v(in_1) = v(in_2)
\end{aligned} \tag{3.4}
$$

証明. 式 (3.4) を理解しやすくするために、否定を取って式 (3.5) を得る。

$$
\begin{aligned}
&\forall in_1, in_2 . \exists t . \\
&\quad (F(in_1, t) = Spec(in_1) \wedge F(in_2, t) = Spec(in_2)) \\
&\quad \vee v(in_1) \neq v(in_2)
\end{aligned} \tag{3.5}
$$

式 (3.4) が UNSAT になることは、式 (3.5) が SAT になることと同値である。式 (3.5) が SAT の時、すべての in_1, in_2 組に対して、同じ値 t で回路が修正できない限りは、パッチの入力 $v()$ の値が異ならねばらないことを表していることが分かる。これは関数の定義と同値であり、式 (3.4) が UNSAT であれば候補 $v()$ がパッチ回路の入力として十分であることになる。

また、式 (3.4) が SAT となるとき、$F()$ はどんな t の値も in_1, in_2 を同時に直せないことになる。パッチの出力は異なる値とならなければならないが、これは式 (3.4) 中の $v(in_1) = v(in_2)$ と矛盾する。よって式 (3.4) が SAT となったときには、$v()$ が不十分であることも示せた。 □

式 3.4 は ∀ を含む QBF 問題である。QBF を解くための手法としては [22, 23] の利用が考えられるが、実装では単に全称記号の部分を展開した。従って、式 (3.6) を用いている。ターゲットの数が N の場合、t の値として 2^N 通りが考えられ、その分展開しなくてはならず問題規模が大きくなってしまう点が欠点である。しかし実験で用いたベンチマーク回路程度の規模であれば、展開した方がアルゴリズムがシンプルになって実行時間が短かった。

$$
\begin{aligned}
&\exists in_1, in_2 . \\
&\quad (F(in_1, (0, \ldots, 0)) \neq Spec(in_1) \vee F(in_2, (0, \ldots, 0)) \neq Spec(in_2)) \\
&\quad \wedge \ldots \wedge \\
&\quad (F(in_1, (1, \ldots, 1)) \neq Spec(in_1) \vee F(in_2, (1, \ldots, 1)) \neq Spec(in_2)) \\
&\quad \wedge v(in_1) = v(in_2)
\end{aligned} \tag{3.6}
$$

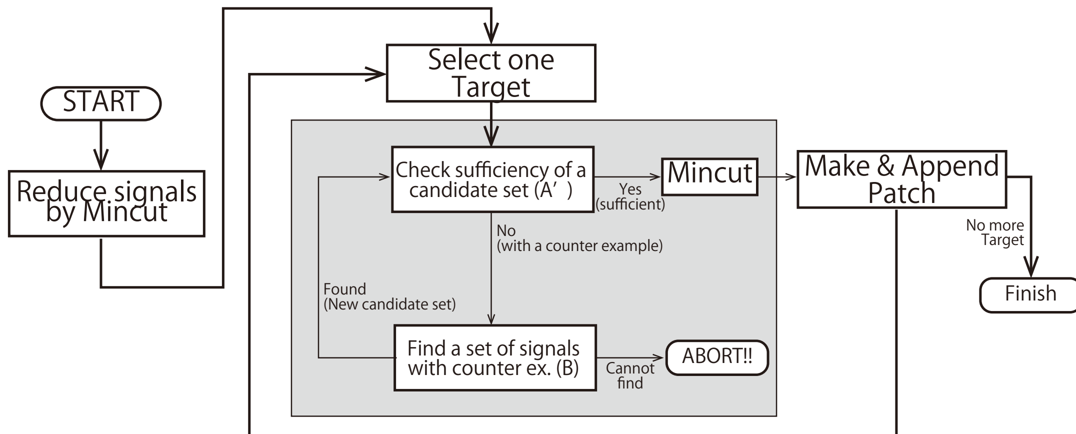


図 3.5: One-by-one method for multiple targets problems

3.5　ターゲット信号の優先順位

3.5.1　複数ターゲットを 1 つずつ解く既存手法

ターゲットが複数ある場合の手法としては、上述の提案手法を除くと [17] にある手法が用いられることが多かった。本節では [17] で述べられている複数ターゲットを取り扱う手法について説明し、次節の提案手法に備えることにする。

この手法は、複数ターゲットを 1 つずつ解くものである。おおまかなフローを図 3.5 に示す。グレーで囲われた部分が、図 3.3 全体と同じで、 1 つのターゲットを直すために必要な信号組を生成している。信号組が見つかった場合にはパッチが生成され、別のターゲットで同じ事が行われる。特に (A') に当たる部分の定式化が以前の式 3.1 とは異なる点に注意されたい。ターゲット信号 t は、選択された 1 つのターゲット信号 t' と残りの信号 t_{rem} に分かれるものとするとする（つまり、$t = (t', t_{rem})$）。そのとき、(A') に当たる式は以下の通りである。

$$\begin{aligned}
&\forall t_{rem} \exists in_1, in_2 \\
&\quad F(in = in_1, t' = 0, t_{rem}) = Spec(in_1) \land F(in = in_1, t' = 1, t_{rem}) \neq Spec(in_1) \\
&\land F(in = in_2, t' = 0, t_{rem}) \neq Spec(in_2) \land F(in = in_2, t' = 1, t_{rem}) = Spec(in_2) \\
&\land v(in_1) = v(in_2)
\end{aligned} \tag{3.7}$$

証明は [17] に譲るが、上式が UNSAT の時に $v()$ は十分であり、SAT の時には不十分である。得られた信号組からパッチ回路を生成し、残りのターゲットについて同様のことを繰り返すと必ず正しいパッチを得ることが出来る。

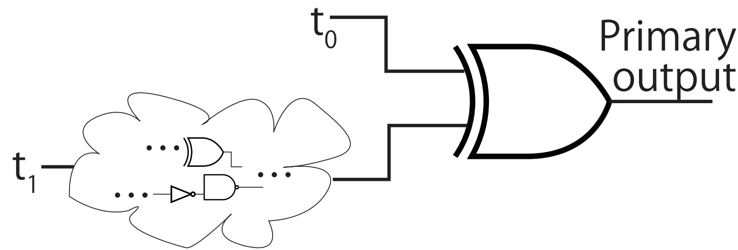


図 3.6: Importance of target priority

3.5.2 ターゲットを取り扱う順番についての提案手法（提案手法）

[17] の手法は必ず正しいパッチ回路が得られるはずであるが、大きな回路では動作しない例が多く実用的ではなかった。筆者が詳細に解析すると、複数ターゲットを 1 つずつ解く手法では、解かれるターゲットの順番が重要であることが分かった。[17] の実装では、扱われるターゲットの順番は信号名のアルファベット順で固定されており、この順番を変更すると解けない問題が解けるようになる例が散見された。

図 3.6 は、ターゲットの順番が実行時間に影響を及ぼす場合の一例である。ターゲットは t_0, t_1 の 2 つであり、t_0 はプライマリ出力のすぐ近くに接続されている。一方で t_1 とプライマリ出力の間には多くの別のゲートがあり、距離が遠い。この場合、t_1 から先に直さないと t_0 の信号組の生成に膨大な時間がかかる可能性が高い。ターゲット信号を 1 つずつ直す場合、はじめのターゲット信号のための入力信号組やパッチ回路には異なる複数の可能性が存在する。もしプライマリ出力の論理をコントロールしやすい位置（距離・レベルの近い位置）にあるターゲットにシンプルなパッチ回路が接続されてしまうと、残されたターゲット信号が複雑な論理を持たなくてはならなくなる可能性がある。複雑な論理を持つ回路はたくさんの入力信号を持つことが多いが、セットカバリング問題として信号選択手法をモデル化した提案手法では、必要な信号組が多くなるほど解を得る時間長くなりやすいという性質がある。従って、ターゲットを直す順番は重要であり、プライマリ出力から遠いターゲットから順にパッチ回路が当てられるべきである。

筆者はターゲット信号の順番を決める手法として Circuit Structural Order (CSO) を提案し、Algorithm 3.1 に示す。プライマリ出力が複数ある場合には、まずプライマリ出力の順番が必要だが、これはコントローラビリティの大きい順としている。なお、"controllability" とは [24] で提案されている CC0（0-コントローラビリティ）,CC1（1-コントローラビリティ）の和である。各プライマリ出力については、距離が遠いものから順に解くものとしている。複数パスがある場合には、最も短い距離を用いて考える物とした。

提案した CSO は実験から得られた発見的手法である。必ず早く解ける保証や、最適解が得られる保証はない。しかし、後述の実験結果によると、すべての回路で最悪の実行時

Algorithm 3.1 Circuit Structure based Order (CSO)

1: $T \leftarrow \phi$ /* T becomes the proposing order */
2: $sortedPO \leftarrow allPOs$, sort by incr. order of controllability
3: **for** each po in $sortedPO$ **do**
4: $tmp \leftarrow$ (targets connected to po)$\setminus T$
5: sort tmp by desc. order of distance to po
6: $T \leftarrow T + tmp$
7: **end for**

間を避けることができ、すべての実行順のなかで早いほうに位置する順番が得られることが分かっている。

3.6 実験

3.6.1 実装とベンチマーク回路

説明した提案手法は 1,000 行程度の C++ で実装した。依存しているソフトウェアは以下の通りである。解きたい問題の等価回路は CNF 式に変換して SAT ソルバで解くが、ABC は変換前の回路を最適化するために使用した。セットカバリング問題の解き方には２種類を用意した。1 つは ILP ソルバを用いて厳密解を求める手法で、もう 1 つは貪欲法によって高速に許容解を求める手法である。厳密手法は実行時間が長いが良い解が得られ、発見的手法は実行時間は短いが解の品質が落ちる可能性がある。集合被覆問題を解くために使う ILP ソルバは CPLEX とした。並列実行可能であるが、最大スレッド数を 4 とした。なお ILP ソルバには 1 回あたり 30 秒の実行時間制限を与えており、ILP ソルバの解も必ずしも最適解ではないが貪欲法よりは良いことが多かった。

- Glucose SAT Solver (v4.0) [7]
- CPLEX ILP Solver (v12.7) [25]
- Boost Graph [26]
- ABC [27]

実行環境は以下の通りである。

- Linux Kernel v4.16
- CPU: Xeon E5-2699v4 (定格 2.2GHz)
- Memory: 512GB

表 3.3: ICCAD'17 contest circuits

NAME	#PIN	#POUT	#Targets	#Gates	# Candidate signals (After mincut)
unit5	450	282	2	24,356	24,746 (57)
unit6	99	128	2	13,826	13,871 (8,647)
unit9	256	245	4	5,846	6,101 (2,835)
unit10	32	129	2	1,579	1,366 (37)
unit11	48	50	8	2,049	2,046 (1,578)
unit14	17	15	12	1,970	1,998 (583)
unit16	417	214	2	2,369	2,267 (1,172)
unit17	136	31	8	2,903	3,038 (797)
unit19	99	128	4	13,345	13,314 (11,059)
unit20	1,874	7,105	4	30,871	23,716 (1,575)

- CPLEX(最大 4 スレッド実行) を除き実行ファイルはシングルスレッド

提案手法の評価に用いたベンチマーク回路は ICCAD'17 の CAD プログラミングコンテスト [21] で用いられたものである。コンテストの設定は本章の問題の定義と同様である。与えられた 20 個の回路のうち複数ターゲット問題は 10 個であったから、実験ではその 10 個だけを用いた。回路の詳細は表 3.3 を参照されたい。与えられた回路は CAD ベンダーの Cadence が作成した例題であり、コンテスト中では 30 分の実行時間制限が与えられた。出題者によると、ツールが満たすべき性能に即した回路規模と実行時間制限である。

3.6.2　最小カット活用手法の評価

第 3.4.2.3 節では最小カットの利用について 2 つの状況を説明した。ここでは序盤に最小カットがどれほどの信号を探索空間から除去できたか確認する。

表 3.3 の最終列では、元の信号数と最小カットによる削減後の信号数が示されている。すべての回路で候補信号数が削減されており、特に unit5 では 99% 以上が不要と判断された。大幅に候補数を削減することによってセットカバリング問題を高速に解くことが出来るようになり、実行時間の短縮に大幅に貢献している。

3.6.3　提案手法の比較

本節では複数の提案手法の比較をする。提案手法には 3 種類あり、1 つは複数ターゲットを同時に取り扱う手法（第 3.4.3 節, At-the-same-time）、もう 1 つは従来手法 [17] をベースにターゲット順を考慮したもの（第 3.5.2 節, One-by-One (CSO)）である。第 3.6.1 節で説明したとおり、セットカバリング問題の解き方として厳密手法 (ILP solver) と発見的手法 (Heuristic solver) がある。本節では、CSO を用いた場合には ILP, Heuristic solver の 2 通りを試した。以上 3 つのバリエーションについての結果を表 3.4 にまとめた。"#Iteration"とは、図 3.3 や図 3.5 内のループの実行回数であり、最終的な解の合計重みは"Weight"に示した。この値は小さい方が良い。実行時間は"Total Time"に示したが、セットカバリングに要した時間も別途示した。

メモリ使用量は別途表 3.5 に示す。解くべき問題のサイズはターゲット数に対して指数的に大きくなるが、実際にターゲット数 12 の unit14 では使用メモリ量が最大 26GB と大きくなっていることが分かる。一方で unit19,20 は数万ゲートの回路だがターゲット数が少ないので、使用メモリ量は高々 2GB 程度だった。複数ターゲットを同時に解く方が、1 つずつ解く手法よりもメモリを多く使う傾向が見られた。ILP ソルバを使うか貪欲法を使うかでメモリ使用量に違いは生じたが、どちらが省メモリとなるかは問題ごとに異なることが分かった。

複数ターゲットを同時に解く手法は他の手法よりも常に良い（もしくは同じ）解が得られており、実行時間が許すならばこちらの手法が最善であることが分かる。一方で実行時間が長く良い結果が得られない場合もある。CSO を使ってターゲットを 1 つずつ解決する場合、多くの場合では同時に解くよりも高速である。セットカバリング問題の解き方による違いは実行時間と解の品質（合計重み）として現れており、厳密手法を使えば良い結果が得られるものの、発見的手法ならばすべての場合の回路を扱えている。使用メモリ量は問題によって異なることから、どの手法が一番省メモリかは一概には判断できない。以上のことから、本研究の最終的な提案手法として以下のフローを提案する。

1. まずはじめに、複数ターゲットを同時に解く手法を ILP ソルバを用いて解く
2. 1. がタイムアウトした場合、1 つずつ解く手法を ILP ソルバを用いて解く
3. 2. もタイムアウトした場合、発見的手法のソルバに変えて解く

3.6.4　ターゲット順 (CSO) の評価

第 3.5.2 節で提案した CSO(Circuit Structural Order) を評価するために、ターゲット数が 4-8 個のベンチマーク回路についてすべての実行順を試して実行時間の違いを計測し

表 3.4: Comparison between “At-the-same-time with ILP solver”, “One-by-One (CSO) with ILP solver”, and "One-by-One (CSO) with Heuristic solver"

	At-the-same-time (ILP)				One-by-One with CSO (ILP)				One-by-One with CSO (Heuristic)			
	#Iteration	Weight	Total Time (s)	Setcover time (s)	#Iteration	Weight	Total Time (s)	Setcover time (s)	#Iteration	Weight	Total Time (s)	Setcover time (s)
Unit5	899	**47**	**11**	0	621	**47**	24	0	418	59	25	0
Unit6	753	N/A	>7,200	6,918[1]	150	**142**	524	22	150	**142**	**504**	0
Unit9	870	**50**	312	303[1]	1,008	358	41	16	1,559	445	27	4
Unit10	647	**135**	122	0	1,124	**135**	62	0	1,147	**135**	**57**	0
Unit11	42,528	N/A	>7,200	6,671	41,102	N/A	>7,200	6,129	2,877	**710**	472	17
Unit14	209	**94**	294	0	105	1,932	474	1	72	1,970	431	0
Unit16	527	**204**	203	191	165	360	12	0	223	468	10	0
Unit17	257	**434**	27	6	205	440	44	0	367	716	39	0
Unit19	14,653	**4,104**	7,204	5,592	2,148	20,258	2,989	151	1,049	16,994	3007	150
Unit20	912	**120**	43	10	2248	302	146	88	1,015	170	48	1

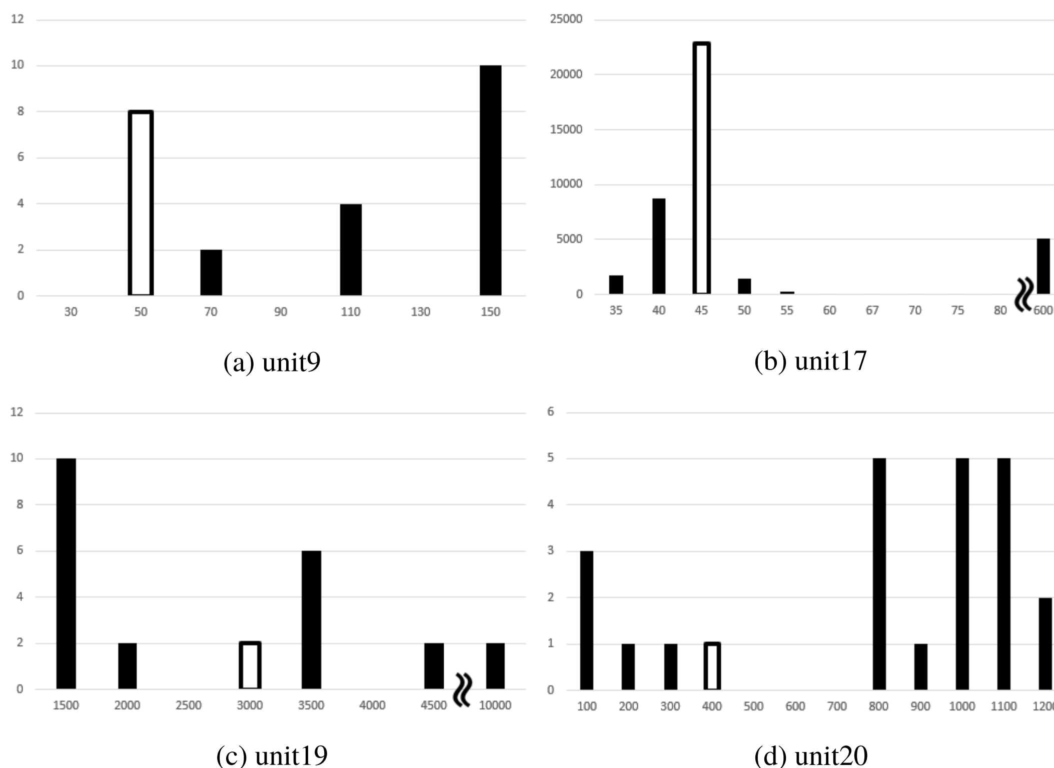
[1] ILP solver reached 20s time limit, and the solution may not be optimal.

表 3.5: Memory Usage (MB)

Name	At-the-same-time (ILP)	One-by-one (ILP)	One-by-one (Heuristic)
unit5	415	279	276
unit6	619	655	1,547
unit9	367	485	506
unit10	80	75	65
unit11	2,485	936	893
unit14	26,564	13,265	13,266
unit16	148	116	115
unit17	2,516	1,311	1,306
unit19	877	811	2,151
unit20	1,907	1,700	1,154

た。度数分布表を図 3.7 に示す。提案手法である CSO を含むバーは白に塗られており、それ以外は黒塗りである。

CSO が含まれる場合でも、必ずしも実行が一番高速であるわけではないことが分かる。しかし、CSO は平均的な実行時間よりは短い実行時間を示しており、何より他に比べて格段に時間がかかってしまうターゲット順を避けることに成功している。以上より、CSO は有用であると考える。



(a) unit9　(b) unit17

(c) unit19　(d) unit20

図 3.7: Target priority and the execution time (X: time, Y: the number of target priorities)

3.6.5　既存手法 [17] との比較

既存手法 [17] は提案手法の中で 1 つずつターゲットを解決する手法と似ているが、CSO を使わない点が異なる。比較結果を表 3.6 に示す。提案手法の結果は表 3.4 内で最も良かったものを採用している。

すべての回路で、信号重みは同じか小さくなっていることが確認できる。信号重みが同じとなった回路については、ILP ソルバによって解が最適解であることが確認されており、これ以上小さく出来ない点に注意したい。unit19 では重みを 1% にまで削減することに成功しており、提案手法が既存手法と比較して高品質な解が得られることが確認できた。

3.7　結論

本章では、組合せ回路を自動修正する際に、直すべきゲートの場所（ターゲット）は分かっているものとして、どの信号を使って直すかを探索する手法について説明した。手法はターゲットが複数ある場合でも動作するものであり、同時に複数ターゲットを解くための定式化を中心として、最小カットを用いた探索空間の削減手法や、 1 つずつ解く従来手法を効率化するためのターゲット順決定手法（CSO）などを提案した。ICCAD’17 CAD

表 3.6: Comparison with the existing research

Name	Ours	[17]	Name	Ours	[17]
unit5	47	47	unit14	94	94
unit6	142	3,942	unit16	204	278
unit9	50	50	unit17	434	434
unit10	135	135	unit19	4,104	490,182
unit11	238	956	unit20	120	120

プログラミングコンテストで用いられたベンチマーク回路で実験を行い、提案手法は既存手法と比較して良い品質の信号組が得られることが確認された。

今後の課題として、解品質を決定するための信号重みの決め方が挙げられる。すべて重みを 1 としたり、修正した場所からの距離に応じた値を設定することが考えられるが、他により良いものがある可能性がある。本研究では実際に得られた解に基づいて再配置配線をし直してはいないので、より現実に即した指標を使ったり、配置配線ツールとの融合が考えられる。

第 4 章

順序回路のための修正信号選択手法

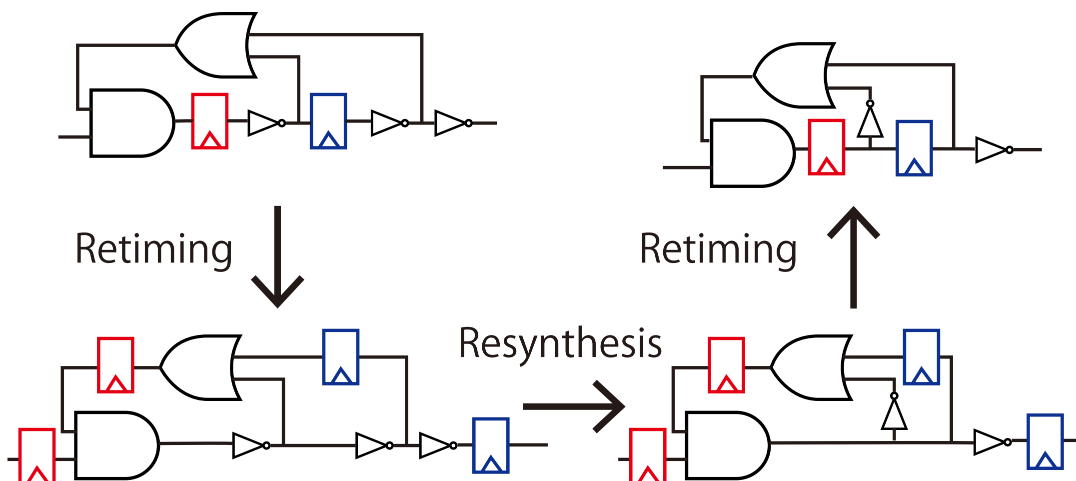


図 4.1: Retiming & Resynthesis

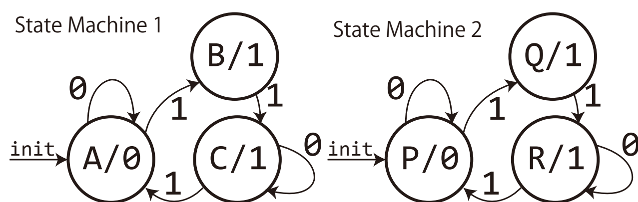


図 4.2: State Correspondence (Moore-type FSM)

4.1　既存研究

第 3 章では組合せ回路を前提とした既存手法を紹介したが、順序回路では唯一 [20] が挙げられる。[20] では修正したい回路には制限があり、仕様回路に Retiming や Resynthesis(図 4.1) などの最適化を行った回路であるという前提がある。これらを適用した回路としていない回路の 2 つを入力とできるが、これ以外の最適化には対応しない。例えば、仕様回路と修正したい回路が全く異なるアルゴリズムで作成された回路の場合、サイクルごとの出力を同じにしたくても修正は出来ない。従って適用範囲が狭いと考える。

4.2　目的

前章では組合せゲートレベル記述を自動修正するために、修正のための適切な信号組を探索する手法を提案した。本章では、順序回路も扱える手法を提案する。

順序回路と組合せ回路との間の大きな違いは状態の有無である。図 4.2 では 2 つの回路がムーア型のステートマシンで表されている。A は P、B は Q、C は R にそれぞれ対応すると考えられる。 2 つの回路を等価にしようとする際に、状態 A を状態 Q や状態 R とま

で等価にしようとすると、小さな変更では正しい回路は得られないことが多い。従って、対応する状態でのみ組合せ回路部分が等価になるようにする必要がある。提案手法では、そのような状態の組合せを網羅するためのアルゴリズムが必要である。

4.3 問題の定義

次の４つ（オプションとしてもう１つ）が入力として与えられる物とする。

- 満たすべき仕様（ゲートレベル記述として与えられる）
- 修正したいゲートレベル記述
- 修正したいゲートレベル記述内で、１つまたは複数の直すべきゲートの場所（ターゲット）
- 2つの回路の初期状態
- （任意）修正したいゲートレベル記述内の、各信号の重み

なお、入力は前章3.3と同様であるから参考にされたい。回路はすべて順序回路であると仮定する。仕様回路と修正したいゲートレベル記述の入出力の対応は分かっているものとする。２つの回路間のFF（フリップフロップ）の数や状態遷移は異なるものであっても良いものとする。回路間のFFの対応関係・状態の対応関係は分かっていない。２つの回路の初期状態は分かっているものとする。仕様には様々な与え方が考えられ、図4.2のような状態遷移も考えられる。しかし、実装では既存の論理合成・検証ツール(abc [27])を活用するために、ゲートレベル記述として与えられるものとした。

この問題では、２つの回路が同じ入力シーケンスに対して同じ出力シーケンスを返すようなパッチ回路を作成したい。このような2回路の関係は"Cycle Accurate"と呼ばれる。本章で提案する手法の目的は前章と同様に、「パッチ回路のための最適な入力組を探す事」である。実験ではすべての重みは"1"としており、入力数を最小にするような探索を行っている。

4.4 例題

提案手法を説明する前に、例題を解いてみることにする。ISCAS'89ベンチマークから、s27という順序回路を用いる。直したい回路は図4.3であり、"t"という信号の論理が分からなくなってしまっている。修正前なので、"i1","f2"はどのゲートにも接続されていない。FFの初期状態は与えられており、すべて0が初期状態である。fn (n=0,1,2)という入出力信号はフリップフロップ(FF)に接続されており、同じ名前の入出力は同じFFの出入力である。元のs27と同じ論理になるようにすることが目標である。

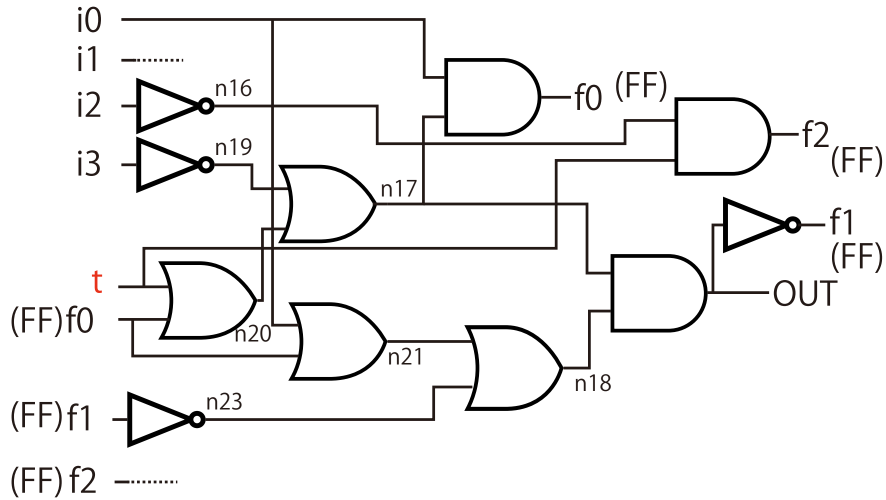


図 4.3: Example circuit to be modified

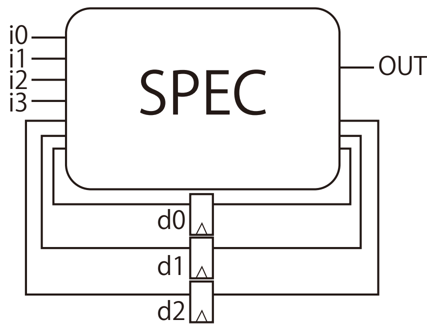


図 4.4: Spec circuit

i0	i1	i3	d0	d1	d2	OUT
0	-	-	0	1	-	0
-	0	1	0	-	0	0
上記以外						1

i0	i1	i3	d0	d2	d0'
0	-	-	-	-	0
-	0	1	0	0	0
上記以外					1

i1	i2	d2	d2'
1	0	-	1
-	0	1	1
上記以外			0

i0	i1	i3	d0	d1	d2	d1'
0	-	-	0	1	-	1
-	0	1	0	-	0	1
上記以外						0

図 4.5: Spec logic

元の s27 の回路構造を 4.4 に示す。直したい回路と同じ数の入出力を持っているが、状態や FF の対応関係は分かっていない。FF 数は偶然同じである。FF の初期状態は与えられており、すべて 0 が初期状態である。論理は図 4.5 に示す。

第 3 章の手法を活用するには、2 つの回路の状態の対応付けを知る必要がある。本節では対応付けを知るための方法を 1 ステップずつ説明する。

まずは初期状態組から始める。この時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。$(i0, i1, i2, i3) = (0, 1, 0, 0)$ の時、t の値に関わらず、OUT の

値は仕様と等価になる。仕様回路の次状態は $(d0, d1, d2) = (0, 0, 1)$ で、修正したい回路が $t = 1$ の時の次状態は $(f0, f1, f2) = (0, 0, 1)$ だから、次はこれを調査することにする。

$(d0, d1, d2, f0, f1, f2) = (0, 0, 1, 0, 0, 1)$ の時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。次に入力 $(i0, i1, i2, i3) = (1, 1, 0, 0)$ を考える。このときも t の値に関わらず OUT の値は仕様と等価であった。仕様回路の次状態は $(d0, d1, d2) = (1, 0, 1)$ で、修正したい回路が $t = 0$ の時の次状態は $(f0, f1, f2) = (1, 0, 0)$ だから、次はこれを調査する。

$(d0, d1, d2, f0, f1, f2) = (1, 0, 1, 1, 0, 0)$ の時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。入力 $(i0, i1, i2, i3) = (1, 1, 1, 0)$ で、t の値に関わらず OUT の値は仕様と等価であった。仕様回路の次状態は $(d0, d1, d2) = (1, 0, 0)$ で、修正したい回路は t の値に関わらず $(f0, f1, f2) = (1, 0, 0)$ である。次はこれを調査する。

$(d0, d1, d2, f0, f1, f2) = (1, 0, 0, 1, 0, 0)$ の時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。どんな入力を与えてもこれまでに得られていない次状態は得られなかった。従って、この状態組以降は調べず、1 つ前の状態に戻る。

$(d0, d1, d2, f0, f1, f2) = (1, 0, 1, 1, 0, 0)$ の時、どんな入力を与えてもこれまでに得られていない次状態組は得られなかった。従って、この状態組以降は調べず、さらに 1 つ前の状態に戻る。

$(d0, d1, d2, f0, f1, f2) = (0, 0, 1, 0, 0, 1)$ の時、入力 $(i0, i1, i2, i3) = (1, 1, 0, 1)$ を考える。$t = 1$ の時しか出力が仕様と等価とならない。次状態は仕様回路は $(1, 0, 1)$、修正したい回路では、$(1, 0, 1)$ である。次はこれを調査する。

$(d0, d1, d2, f0, f1, f2) = (1, 0, 1, 1, 0, 1)$ の時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。一方で、どんな入力を与えてもこれまでに得られていない次状態組 $(d0, d1, d2, f0, f1, f2)$ は得られなかった。従って、この状態組以降は調べず、1 つ前の状態に戻る。

$(d0, d1, d2, f0, f1, f2) = (0, 0, 1, 0, 0, 1)$ の時、どんな入力を与えてもこれまでに得られていない次状態組は得られなかった。従って、この状態組以降は調べず、1 つ前の状態に戻る。

初期状態組に戻ってきた。$(i0, i1, i2, i3) = (1, 0, 1, 0)$ の時、$t = 0$ の時のみ OUT の値は仕様と等価になる。仕様回路の次状態は $(d0, d1, d2) = (0, 1, 0)$ で、修正したい回路が $t = 0$ の時の次状態は $(f0, f1, f2) = (0, 1, 0)$ だから、次はこれを調査することにする。

$(d0, d1, d2, f0, f1, f2) = (0, 1, 0, 0, 1, 0)$ の時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。入力 $(i0, i1, i2, i3) = (1, 1, 0, 0)$ を考える。このときも t の値に関わらず OUT の値は仕様と等価であった。仕様回路の次状態は $(d0, d1, d2) = (0, 1, 1)$ で、修正したい回路が $t = 1$ の時の次状態は $(f0, f1, f2) = (0, 1, 1)$ だから、次はこれを調査する。

表 4.1: Reachable state pairs (Incorrect)

d0	d1	d2	f0	f1	f2
0	0	0	0	0	0
0	0	1	0	0	1
1	0	1	1	0	0
1	0	0	1	0	0
1	0	1	1	0	1
0	1	0	0	1	0
0	1	1	0	1	1

$(d0, d1, d2, f0, f1, f2) = (0, 1, 1, 0, 1, 1)$ の時、どんな入力でも適切な値を t に与えれば、必ず修正出来ることを確認した。一方で、どんな入力を与えてもこれまでに得られていない次状態組は得られなかった。従って、この状態組以降は調べず、1 つ前の状態に戻る。これ以降、新しい次状態組を得ることは出来なかった。これまでに得られた適切な状態組を表 4.1 に記す。作業中に、t の値を 0 にするか 1 にするか自由に選択できる場合があった。この選択によって異なる表が得られることがある点に注意されたい。

次に、これらの状態だけになるように制限を加えて第 3 章の手法を応用する。以下の式は、第 3 に記された、信号組 $v()$ が修正に十分かどうかを確認する式であり、UNSAT となれば十分である。$F()$ や $Spec()$ はここでは組み合わせ回路である。

$$\begin{aligned}&\exists in_1, in_2.\forall t.\\&\quad (F(in_1, t) \neq Spec(in_1) \vee F(in_2, t) \neq Spec(in_2))\\&\quad \wedge v(in_1) = v(in_2)\end{aligned}$$

順序回路の場合は入出力として状態が追加されるから、仕様回路の入力状態 st_{SPEC}、修正したい回路の入力状態 st_F が、表 4.1 に含まれるという条件を追加して解く。上式の 2 行目では回路が仕様と正しくなくなる条件を書いているが、ここに次状態が表 4.1 に含まないという条件も追加すればよい。詳細な定式化は後述するが、式 4.3 を使えば良い。この式を繰り返し用いることによって、修正のための信号組が得られる。

t の値の選択によってはさまざまな表が得られる可能性に言及したが、実際に表 4.1 を到達可能状態組とすると修正のための信号組が得られない。$(1, 0, 1, 1, 0, 0), (1, 0, 0, 1, 0, 0)$ の 2 つは、異なる値を t に与えないと回路が修正できないことが式 4.3 から分かった。しかし、$(f0, f1, f2) = (1, 0, 0)$ と同じ入力状態であるから、t が異なる値をとるような関数を作ることは不可能である。いずれかを、登場してはならない状態組として問題を解き直す必要がある。このような状態組を「到達不可」と表現し、以下で問題を解き直す。

表 4.2: Reachable state pairs (Correct)

d0	d1	d2	f0	f1	f2
0	0	0	0	0	0
0	0	1	0	0	1
1	0	0	1	0	0
1	0	1	1	0	1
0	1	0	0	1	0
0	1	1	0	1	1

$(1,0,0,1,0,0)$ を到達不可として同じ問題を解くと、以下のようになる。まず初期状態組 $(d0,d1,d2,f0,f1,f2)=(0,0,0,0,0,0)$ の時、入力 $(i0,i1,i2,i3)=(0,1,0,1)$ を考える。$t=1$ の時しか出力が仕様と等価とならない。次状態は仕様回路は $(0,0,1)$、修正したい回路では $(0,0,1)$ である。$(d0,d1,d2,f0,f1,f2)=(0,0,1,0,0,1)$ の時、入力 $(i0,i1,i2,i3)=(0,1,0,1)$ を考える。$t=1$ の時しか出力が仕様と等価とならない。次状態は仕様回路は $(1,0,1)$、修正したい回路では $(1,0,1)$ である。$(d0,d1,d2,f0,f1,f2)=(1,0,1,1,0,1)$ の時、入力 $(i0,i1,i2,i3)=(1,1,1,1)$ を考える。t の値に関わらず出力は仕様と等価な値である。次状態は仕様回路は $(1,0,0)$、修正したい回路では t の値に関わらず $(1,0,0)$ である。しかし、$(d0,d1,d2,f0,f1,f2)=(1,0,0,1,0,0)$ は認めないことにしているので、正しくない。t の値に関わらず $(1,0,1,1,0,1)$ は次状態に $(1,0,0,1,0,0)$ を持つから、$(1,0,1,1,0,1)$ もまた到達不可となる。同様にして $(0,0,1,0,0,1),(0,0,0,0,0,0,)$ も到達不可となり、初期状態組が到達不可となってしまう。従って、$(1,0,0,1,0,0)$ を到達不可として考えると、回路のデバッグができない。

もう一方の反例 $(1,0,1,1,0,0)$ を到達不可とした場合、表 4.2 のような到達可能状態組が得られる。このときには、第 3 章の手法より $t=OR(i0,f2)$ とすれば良いことが分かり、正しい回路が得られた。

このように、本提案手法では 1 つずつ到達可能な状態組をたどっていくことで、全到達可能状態組を網羅する。ただし得られた状態組のうち、同時には実現出来ないものが含まれる場合がある。その場合にはどちらかを反例として再度同じことをすることとなる。

フリップフロップ数が仕様回路と修正したい回路で合わせて 6 つあるので、例題で考えられる状態組数は計 $2^6=64$ 通りである。最悪の場合、すべての状態組を網羅する必要があり、フリップフロップ数が数十の大きな回路では膨大な実行時間が必要となる可能性がある。

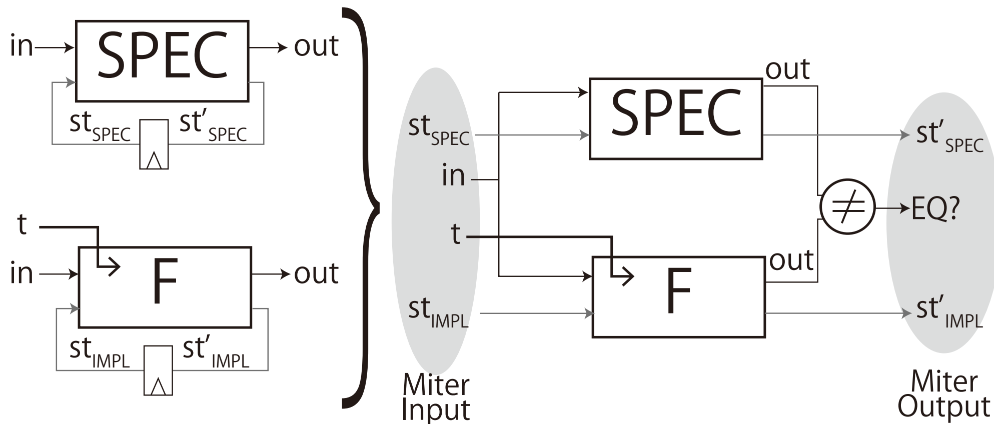


図 4.6: Miter Circuit

4.5 提案手法

まずいくつかの記号の定義をする。フリップフロップから仕様回路への入力を st_{spec} と表記し、修正したい回路の場合には st_{impl} とする。回路の出力は入力値と状態値から決まるから、仕様回路の出力は $out = Spec(in, st_{spec})$、次状態は $st'_{spec} = Spec_{ST}(in, st_{spec})$ と表記する。修正したい回路の場合にはターゲットの値を定める必要があり、出力は $out = F(in, t, st_{impl})$、次状態は $st_{impl} = F(in, t, st_{IMPL})$ と表記する。初期状態はそれぞれ $Init_{spec}, Init_{impl}$ と表記する。初期状態はそれぞれ予め分かっているものとする。

提案手法では、図 4.6 のようなマイター回路を考える。仕様回路と修正したい回路の間で対応する状態 (st_{spec}, st_{impl}) を状態組と呼び、到達可能な状態組を格納するスタックを Reachable とする。逆に、到達すべきでない状態組は Unreach に格納する。初期状態組は到達可能であるべきだから、まず初めは Reachable に挿入される。提案手法はこのような状態組に基づいているから、1 対 1 対応する状態組だけでなく、1 対多や多対多の関係性も取り扱うことが出来る。このことについては実験でも確認する。

提案手法の実行フローを図 4.7 に示す。手法は以下の 2 つのステップを繰り返し行うことで修正のための信号組を生成する。

- 【作業１】到達可能状態組の候補を生成する
- 【作業２】得られた到達可能状態組を用いて信号組を生成する

誤りのある回路を仕様と等価にするためには、初期状態 $(INIT_{spec}, INIT_{impl})$ から始めて到達可能状態組だけを考えれば良い。2 回路の間で対応しない状態や到達しない状態まで考えようとすると、回路は修正出来ない可能性が高いから、この作業は必須である。以上のことは【作業１】で行われる。【作業２】では得られた到達可能状態組をもとに修正のための信号組を生成するが、第 3 章で用いた手法を利用する。注意したいのは、【作業

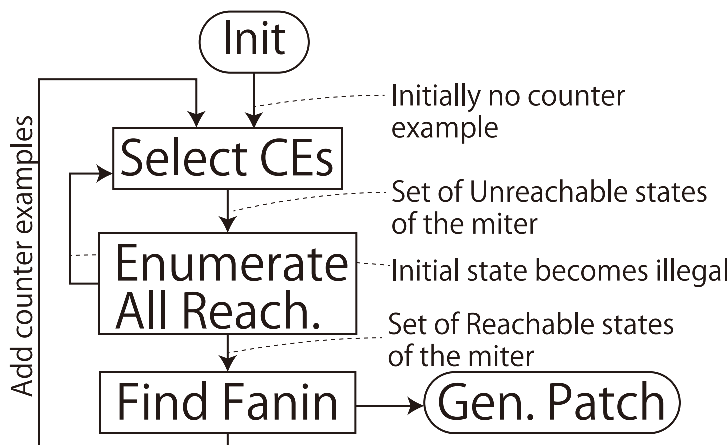


図 4.7: Overall flow of the proposed method

１】で得られる到達可能状態組にはいくつか異なる解があるという点である。中には【作業２】で信号組が得られないことがあり、その場合には同時に対応できない 2 状態組が得られる。２つの対応出来ない状態組 $(st_{1spec}, st_{1impl}), (st_{2spec}, st_{2impl})$ は、同時に修正出来るようなターゲット関数を作ることが出来ない。従って、２つのうちのいずれか一方を予め到達してはいけない状態組としたうえで再度【作業１】を行ばよい。このような状態組は「到達不可」と呼ぶ。以上のことを繰り返せば、正しい信号組が生成できる。以上の内容を擬似コードを Algorithm 4.1 にも示した。

4 行目の genUnreach() 関数は、対応できない状態組のペア CE1,CE2 のいずれかを選んで到達不可状態組を決める関数である。初回ループでは、この関数は空集合を返す。5 行目の checkState() 関数が【作業１】である。初期状態組から始めて到達可能状態組の候補すべてがスタック Reachable に格納される。計算の際には、事前に与えられた到達不可組を除外するようにしている。詳細は次節で紹介する。もし計算の結果 Reachable のサイズが 0 の場合、初期状態組も含めて到達不可であることになり、反例選択のやり直しを行う。9 行目では【作業２】が行われており、適切な信号組が得られない場合には対応出来ない２つの状態組 $(st_{1spec}, st_{1impl}), (st_{2spec}, st_{2impl})$ が得られ、それぞれ CE1,CE2 に格納される。

genUnreach() 関数の擬似コードを Algorithm 4.2 に示す。分かりやすい模式図として図 4.8 も参照されたい。(a) は新たな状態組 CE1,CE2 が追加された場合、(b) はこれまでの選択が誤っていた場合を示し、以下で説明する。初めは 2 行目にあるように空集合を返す。Algorithm 4.1 の 13-14 行目で新たな CE1,CE2 が追加されたときには、Algorithm 4.2 の 5 行目の条件が True となる。この関数はデフォルトでは CE1 にある状態組を選択するようになっているので、CE1 の内容が新たに追加されることになる。この動作は図 4.8(a) に対応する。

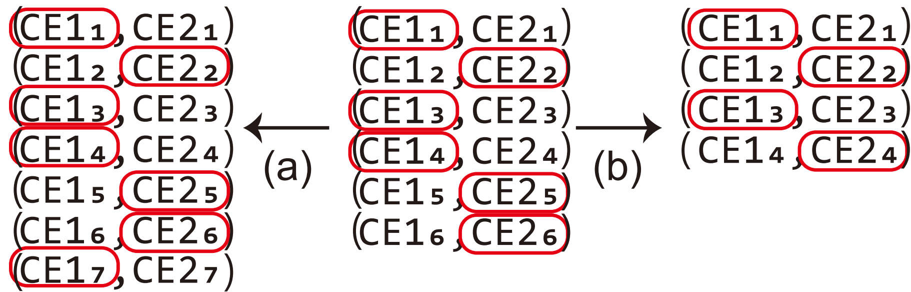


図 4.8: Counter examples selection

Algorithm 4.1 Overall Flow of Proposing Method

1: Reachable[] $\leftarrow \phi$; Unreach[]$\leftarrow \phi$
2: CE1[] $\leftarrow \phi$; CE2[] $\leftarrow \phi$ //These are stacks (vectors).
3: **while** True **do**
4: 　Unreach $\leftarrow$ $genUnreach$(CE1, CE2, Unreach)
5: 　checkState(($Init_{spec}, Init_{impl}$)) // Job1
6: 　**if** $size$(Reachable) = 0 **then**
7: 　　**continue**
8: 　**end if**
9: 　hasSolution $\leftarrow$ $findFanin$(Reachable) // Job2
10: 　**if** hasSolution **then**
11: 　　getPatch(); **break**
12: 　**else**
13: 　　CE1.push((st_{1spec}, st_{1impl}))
14: 　　CE2.push((st_{2spec}, st_{2impl}))
15: 　**end if**
16: **end while**

もし Algorithm 4.1 の 2 行目の条件が True になって【作業 1】に失敗した場合には、到達不可状態組の選択が不適切だったことになる。この場合には、Algorithm 4.2 の 5 行目の条件が False となり、もっとも直近の CE1 の選択が CE2 に変更される。これを行っているのが 8-13 行目であり、図 4.8(b) に対応している。これによって、適切な到達不可状態組が得られるまですべての選択肢が試行できる。

Algorithm 4.2 genUnreach

Input: CE1[], CE2[], Current[] //These are stacks (vectors).

Output: Result[]

```
1: Result[] ← φ
2: if size(CE1) = 0 then
3:     return Result
4: end if
5: if size(Current) is NOT size(CE1) then
6:     Result ← Current + CE1.top()
7: else
8:     while Current.top() is CE2.top() do
9:         CE1.pop(); CE2.pop()
10:        Current.pop()
11:    end while
12:    Result ← Current
13:    Result.top() = CE2.top()
14: end if
15: return Result
```

4.5.1 （作業１）到達可能状態組の生成

本節では、checkState() 関数を説明する。この関数は再帰関数であり、初期状態組から始めて Reachable と Unreach を繰り返し更新する。擬似コードを Algorihtm 4.3 に示した。図 4.9 も参考にされたい。

詳細の説明の前に、"Legal"の定義を確認する。本手法では状態組 (st_{spec}, st_{impl}) について、以下の式 (4.1) が SAT となると Legal、UNSAT となると Illegal であるとする。

$$\forall in.\exists t.Spec(in, st_{spec}) = F(in, t, st_{impl}) \tag{4.1}$$

Legal であるとは、調査中の状態組 (st_{spec}, st_{impl}) に限れば、適切なターゲットの値 t を用いればすべての入力 in で回路が修正可能であるということである。逆に Illegal であるとは、どんな t の値でも修正出来ない in があるということであり、Illegal な状態組は回路を修正出来ない。Illegal な状態組は Unreach に含まれるが、Legal な状態組が Reachable に含まれるか Unreach に含まれるかは分からない点に注意したい。

例えば図 4.9 で状態組 $current_2$ は Legal だが、ある入力 in_C を与えたとき、どんな t の値でも出力を仕様と等価に出来ない。次状態が常に Illegal な状態組となるので、$current_2$

は Unreach に入る。一方 $current_1$ では、入力 in_A の際には次状態組に Legal なものと Illegal なものがある。t の値によって、同じ状態組・同じ入力値でも複数の次状態組が考えられる。各入力値について、次状態組候補のうち少なくとも 1 つが Reachable であれば、調査中の状態組も Reachable であるとする。in_B については後で説明する。

式 (4.1) を解くに当たっては、ターゲット t の数 (ビット幅) が入力ビット数にくらべてそれほど多くないと仮定すると、否定を取った式 (4.2) を用いることも出来る。式 (4.2) が SAT となると Illegal、UNSAT となると Legal である。ただし、式 (4.2) には依然として $\forall$ が残ってしまう。本研究では $2^{size(t)}$ 回だけ式を展開し、式 (4.3) を SAT ソルバで解くようにした。

$$\exists in.\ \forall t.Spec(in, st_{spec}) \neq F(in, t, st_{impl}) \tag{4.2}$$

$$\Updownarrow$$

$$\begin{aligned} \exists in.\ & Spec(in, st_{spec}) \neq F(in, (0, 0, \ldots, 0), st_{impl}) \\ & \wedge \ldots \\ & \wedge Spec(in, st_{spec}) \neq F(in, (1, 1, \ldots, 1), st_{impl}) \end{aligned} \tag{4.3}$$

checkState() 関数について、Algorithm 4.3 の詳細を以下に記す。この関数の入力はある状態組 $current = (st_{spec}, st_{impl})$ であり、bool 値 (到達可能かどうか) を返す。まず 6 行目で Legal かどうかが確認される。もし Illegal の場合、$current$ は Unreach に格納され、関数は 23 行目で False を返す。Legal の場合には到達可能かどうかの判断はこの時点では出来ないので、9-18 行目で、すべての入力について少なくとも 1 つの次状態が Reachable になることが確認される。

ある入力 in について t の値によっては複数の次状態が存在することは前述の通りだが、その次状態のうちのいくつかが既に Reachable 内に入っていることがある。このような入力 in は調査の必要がないので、9-18 行目中では除外して考えたい。具体的には、図 4.9 中の入力 in_B では、次状態の 1 つが初期状態である。このような in_B は追加の調査は必要ない。従って、Reachable に入っている次状態を持たない入力だけを調査するために式 (4.4) を利用する。式 (4.4) は $\forall$ を含むので、実装では式 (4.5) を用いている。Algorithm 4.3 中の 10-14 行目では少なくとも 1 つの次状態組が Reachable であることが再帰的に確認されている。

以上の計算によって、すべての到達可能状態組が得られることになる。

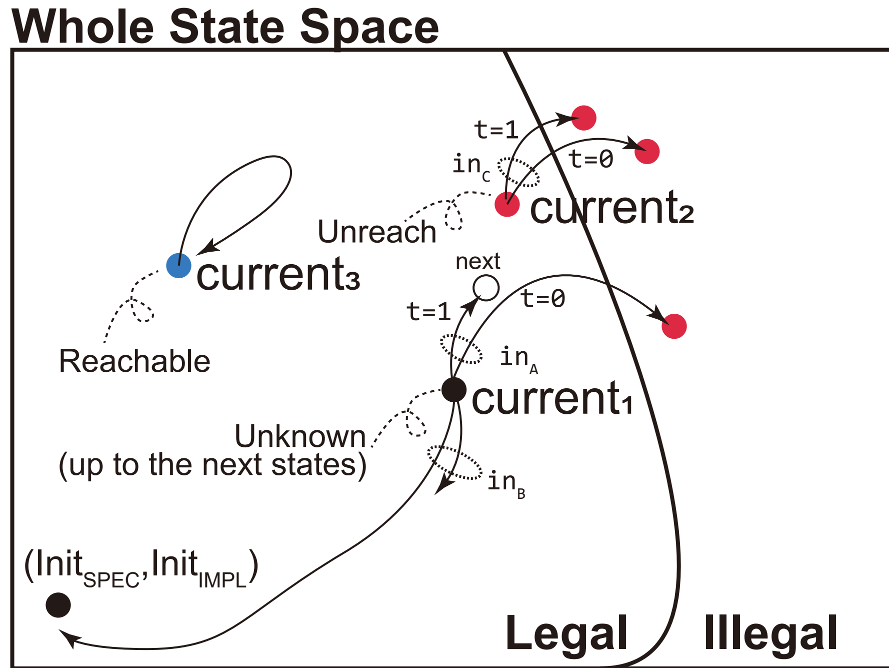


図 4.9: Search for the reachable state pairs

$$
\begin{aligned}
&\exists In.\forall t.\\
&Spec(In, st_{spec}) = F(In, t, st_{impl})\\
&\rightarrow (Spec_{ST}(In, st_{spec}), F_{ST}(In, t, st_{impl})) \notin R
\end{aligned}
\tag{4.4}
$$

$$
\begin{aligned}
&\Updownarrow\\
&\exists In.\\
&Spec(In, st_{spec}) = F(In, (0, \ldots, 0), st_{impl})\\
&\rightarrow (Spec_{ST}(In, st_{spec}), F_{ST}(In, (0, \ldots, 0), st_{impl})) \notin R\\
&\wedge \ldots \wedge\\
&Spec(In, st_{spec}) = F(In, (1, \ldots, 1), st_{impl})\\
&\rightarrow (Spec_{ST}(In, st_{spec}), F_{ST}(In, (1, \ldots, 1), st_{impl})) \notin R
\end{aligned}
\tag{4.5}
$$

4.5.2 （作業２）修正のための信号組生成

回路の修正のための信号組は、第 3 章の手法を活用して得られる。順序回路に対応するために、式 (3.4) を改変し、式 (4.6) のようにする必要がある。

定理 3. 以下の式 *(4.6)* が *UNSAT* となれば、信号組 *v()* は修正に十分である。*SAT* となれ

Algorithm 4.3 checkState

Input: $current$ // a state pair (st_{spec}, st_{impl}) to be checked

Output: True or False // Reachable or Unreach

```
1: if current ∈ Reachable then
2:     return True
3: else if current ∈ Unreach then
4:     return False
5: end if
6: isLegal ← Formula (4.3) is UNSAT
7: if isLegal then
8:     Reachable.push(current) // Assuming current is reachable
9:     while Formula (4.5) is SAT do
10:        for each possible next state pair next do
11:            if checkState(next) then
12:                goto line 9
13:            end if
14:        end for each
15:        Remove all items after current from Reachable
16:        Unreach.push(current)
17:        return False
18:    end while
19:    return True
20: else
21:    Remove all items after current from Reachable
22:    Unreach.push(current)
23:    return False
24: end if
```

ば、不十分である。

$$
\begin{aligned}
&\exists in_1, st_{1spec}, st_{1impl}, in_2, st_{2spec}, st_{2impl}.\forall t.\\
&\quad (st1_{spec}, st1_{impl}) \in R \wedge (st2_{spec}, st2_{impl}) \in R\\
&\quad \wedge [F(in_1, t, st_{1impl}) \neq Spec(in_1, st_{1spec})\\
&\qquad \vee (F_{ST}(in_1, t, st_{1impl}), Spec_{ST}(in_1, st_{1spec})) \notin R\\
&\qquad \vee F(in_2, t, st_{2impl}) \neq Spec(in_2, st_{2spec})\\
&\qquad \vee (F_{ST}(in_2, t, st_{2impl}), Spec_{ST}(in_2, st_{2spec})) \notin R]\\
&\quad \wedge v(in_1) = v(in_2)
\end{aligned} \tag{4.6}
$$

証明. 上式は式 (3.4) を元にしているから、そちらの証明も参照されたい。式 (4.6) では全７行のうち、2,4,6 行目が追加されている。２行目は、探索空間を与えられた Reachable 内に絞る制約式である。到達不可能な状態組を探索する必要がないため、このようになっている。3,5 行目は、仕様と等価になるという条件の否定を表している。順序回路の場合、修正すべき回路が仕様と等価になるには以下の２つが両方満たされなくてはならない。

- 出力が仕様と等しい
- 次状態が到達可能状態組に含まれる

１つ目の条件が 3,5 行目に当たる。２つ目の条件は組合せ回路には存在しないので、その部分を追加したのが 4,6 行目である。 □

式 (4.6) が SAT になると、$in_1, st_{1impl}, in_2, st_{2impl}$ が得られるから、表 3.1 のような表を作ることが出来る。従って、修正のための信号組の生成は第 3 章と同様に行える。

注意したいのは、$in_1 = in_2$ と $st_{1impl} = st_{2impl}$ が同時に成り立つような反例が得られた場合である。この場合、表 3.1 中の最後の行がすべて 0 となり、次の候補信号が得られない。これは、【作業１】で得られた到達可能状態組が不適切であり、その反例として CE1:(st_{1spec}, st_{1impl}), CE2(st_{2spec}, st_{2impl}) が得られたことになる。もしこの CE1,CE2 が同時に Reachable の中に含まれると、回路が修正出来ないことになる。従って、いずれか１つは Unreach に予め含まれるべきである。

なお、式 (4.6) は ∀ を含む QBF 問題であるから、実装では SAT ソルバで解くために $2^{size(t)}$ 回だけ展開している。QBF ソルバを用いて問題を解くことも出来るが、ターゲット数は高々 2 つ程度だと見積もり、∀ 部分を展開した方が合計実行時間は短くなると考えた。

4.5.3 計算量について

提案手法は到達可能な状態組を 1 つずつ網羅する手法である。仕様回路のフリップフロップ数を m、修正したい回路のフリップフロップ数を n とすると、合計 2^{m+n} 通りの状態組が存在する。従って最悪の場合では、その分の checkState() 関数を実行する必要がある。一般に順序回路には状態遷移に関わるフリップフロップがあり、到達可能な状態組数は 2^{m+n} よりは小さくなることが一般的である。実際に次節の実験結果では、全体の 1% 程度しか到達しない場合もあった。

しかし、たとえ 1% でも膨大になり得るので、探索範囲を削減する必要がある。フリップフロップには 2 種類が存在し、1 つは前述の状態遷移に関わるフリップフロップ、それ以外はすべての値を取ると考えられるデータパス系フリップフロップである。実験の中には、データパス系のフリップフロップについては仕様回路と修正したい回路の間で対応が

取れているものと考え、すべての値を取り得るものとした場合がある。つまり、データパス系のフリップフロップはプライマリ入出力と同等に扱って実験を行った。実験で用いたITC99 ベンチマーク回路では、状態遷移系のフリップフロップには"stato"などの名前がついていることが多く、簡単に 2 種を分類することができた。

また、到達可能な状態組や不可能な状態組の一部が既知の場合には、その分だけ計算が高速化する可能性にも注意したい。もし到達可能な状態組が分かっている場合、checkState() の回数を減らしたり式 (4.5) で該当する入力値を減らすことが出来る。到達不可能状態組が分かっていれば、checkState() でループに入る前に False を返すことが出来、こちらも効率的になる。合成ツールなどで少しでもこのような情報があれば活用すべきである。ただし、誤った回路設計で得られたこのような情報も、誤っている可能性がある。この場合、必要以上に大きなパッチ回路が得られたり、修正出来るはずの回路が修正出来ない可能性があるから注意したい。

4.6　実験

4.6.1　実装とベンチマーク回路

説明した提案手法は 1,000 行程度の C++ で実装された。およそ第 3 章と同様であるが、依存しているソフトウェアのバージョンが変わっているものがある。以前の実装を拡張する形で実装したが、不要なコードを除去したため C++1,000 行程度である点に変わりは無い。

第 3 章ではセットカバリング問題の解き方に 2 種類を用意したが、本章では ILP ソルバによる厳密手法のみを用いた。本章の実験では到達可能状態組の生成に多くの時間が掛かっており、パッチの生成にはさほど時間が掛かっていないため、セットカバリング問題の解き方による実験結果への影響は小さい。ILP ソルバには 1 回あたり 30 秒の実行時間制限を与えた。

- Glucose SAT Solver (v4.0) [7]
- CPLEX ILP Solver (v12.8) [25]
- Boost Graph [26]
- ABC [27]

実行環境は以下の通りである。

- Linux Kernel v4.19
- CPU: Xeon E5-2699v4 (定格 2.2GHz)
- Memory: 512GB

表 4.3: Statistics of ITC99 circuits

Name	# In	# Out	# Gates	# FF	# FF (st)	comment
b01	2	2	56	5	3	
b02	1	1	25	4	3	
b03	4	4	150	30	2	
b04	11	8	569	66	2	
b05	1	36	603	34	3	
b06	2	9	48	9	3	
b07	1	8	418	44	3	
b08	9	4	167	21	2	
b09	1	1	150	28	2	
b10	11	6	204	17	6	
b11	7	6	512	30	4	
b13	10	10	290	48	11	
b14	32	54	6,379	215	3	
b15	35	70	7,080	416	7	
b17	36	97	21,771	1,314	21	3*b15
b18	35	30	73,202	3,012	48	2*b14+6*b15
b19	45	40	135,292	6,026	96	4*b14+12*b15
b20	32	22	13,784	430	6	2*b14
b21	32	22	13,869	430	6	2*b14
b22	32	22	20,891	613	9	3*b14

- CPLEX(最大 4 スレッド実行) を除きシングルスレッド

実験には ITC99 ベンチマーク回路を用いた。詳細を表 4.3 に示す。このベンチマーク回路は人間にも読みやすい合成前の VHDL として与えられる順序回路である。本実験ではセットカバリング問題を解くに当たっての信号の重みはすべて 1 とする。従って、信号組の信号数がなるべく小さくなるような解が生成される。

4.6.2　ランダムに設定したターゲット信号の修正

ひとつの VHDL ファイルから、以下の 21 個の設計ファイルを作成した。

- そのまま論理合成したファイル。仕様として使用
- 論理合成して、その中からランダムに 1 ゲート選んで種類を変えたもの。選択したゲートの 1 出力をターゲットとして指定する。この作業を 5 回行い、修正すべき回路を 5 つ作った
- 論理合成して、その中からランダムに 2 ゲート選んで種類を変えたもの。選択したゲートの 2 出力をターゲットとして指定する。この作業を 5 回行い、修正すべき回路を 5 つ作った
- 状態遷移のエンコーディングを変更したものを論理合成して、その中からランダムに 1 ゲート選んで種類を変えたもの。選択したゲートの 1 出力をターゲットとして指定する。この作業を 5 回行い、修正すべき回路を 5 つ作った
- 状態遷移のエンコーディングを変更したものを論理合成して、その中からランダムに 2 ゲート選んで種類を変えたもの。選択したゲートの 2 出力をターゲットとして指定する。この作業を 5 回行い、修正すべき回路を 5 つ作った

1 つ目のファイルは仕様回路である。残りは修正すべき回路である。ゲートをランダムに選び、その出力をターゲットとして考えた。はじめの 10 個は仕様回路とトポロジーが似た回路である。そのほかに、提案手法が異なるトポロジーを持つ回路でも適用可能であることを示すために、状態遷移のエンコーディングを変えた回路も用意した。ITC99 ベンチマーク回路の VHDL ファイルでは、状態遷移に関するフリップフロップ名が"stato"などから始まるため、簡単に特定してエンコーディングを変えることが出来た。仕様回路の合成後のゲート数などは表 4.3 に示した。修正したい回路のゲート数は、ゲートの種類を変えて問題を作っているので仕様回路と変わらない。

提案手法の実験結果を表 4.4,4.5 に示す。6 列目が図 4.7 のループの実行回数である。その中で最後に実行されたループの実行時間の内訳が 3,5 列目であり、それぞれ到達可能状態の計算・信号組の生成に当たる。実行可能状態数は 2 列目、得られた信号組の中の信号数を 4 列目に示す。全体の実行時間は最終列である。

エンコーディングによって分けて結果を示したが、両者に大きな違いは見られなかった。多くの回路では図 4.7 内のループの実行回数は 1 回で、正しい到達可能状態組が初めから得られたことを示している。到達可能状態組が得られた回路は数百ゲート程度であったから、信号組の生成には時間がかからず、到達可能状態組の網羅に最も時間がかかっている。すべての到達可能状態組を網羅するには、状態組数が数千で 1 分ほど、3 万で 1 時間ほどかかっている。b08 の仕様回路は 21 個のフリップフロップを持ち、修正したい回路は 22 個である。従って 2^{43} 個の状態組が存在するが、実際に到達可能状態組となったのは 29,186 個だけだった。このように、多くの場合では到達可能状態組は全状態組よりは少ないことが分かる。従って提案手法は、特に状態遷移フリップフロップが多い場合に

表 4.4: Buggy Circuits with Same Encoding (average of 10 problems)

Name	In Final Loop				# Iter	Total Time (s)
	# Reachable Stat. Pairs	Reach. Stat. Calc. Time (s)	# Iter. for fanin select.	Fanin select. Time (s)		
b01	8	0.40	13	0.46	1	0.97
b02	7	0.34	5	0.41	1	0.87
b03	2058	8.4	7	37.5	3	38
b04	N/A	>3h	N/A	N/A	1	>3h
b05	70	1.2	19	14	1	14
b06	7	0.35	6	0.54	1	0.97
b07	87	0.35	5	4.1	1	4.8
b08	29,186	32min or >3h	163	3min	1	35min or >3h
b09	N/A	>3h	N/A	N/A	1	>3h
b10	4465	65	55	132	5	300
b11-b22	N/A	>3h	N/A	N/A	1	>3h

は大きな回路にも適用可能である。

メモリー使用量の平均を表 4.6 に示した。この結果は表 4.4,4.5 の実験の平均である。２つの間に大きな差は見られなかったから、メモリ使用量についてはまとめて平均を示した。b11 以降の 3 時間でタイムアウトした問題でも、メモリ使用量は高々 200MB 程度と小さかった。3 時間では数万から十数万の状態組数を扱った。従って、本手法のボトルネックはメモリ使用量ではないことが分かる。以降の実験結果ではメモリ使用量は示さないが、いずれもメモリ使用量は同等程度であった。注意したいのは、タイムアウトした問題の中には状態数が爆発的に多い場合も考えられ、長時間かけて実際に解くとメモリ使用量も膨大であることは考えられる。

b04,b09,b11-b22 はかなり多くのフリップフロップを持っており、提案手法は実行時間が 3 時間を超え終了しなかった。すべて、到達可能状態組を網羅するための時間が掛かりすぎ、タイムアウト（3 時間）したことが原因である。30000 程度の状態組数は扱えていることを考えると、$2^{1}5 = 32768$ であるから、15 個程度の FF を含む問題ならば概ね扱うことが出来ると考える。タイムアウトした問題は仕様回路だけで数十から数百の FF を含んでおり、探索空間が広すぎたと考えられる。

表 4.7,4.8 は得られた信号組の中の信号数をまとめたものである。0, 1, 2, 3, 4, 5+ は信号数であり、1 ターゲット回路・2 ターゲット回路それぞれについて示した。たとえば、2

表 4.5: Buggy Circuits with Different Encoding (average of 10 problems)

Name	In Final Loop				# Iter	Total Time (s)
	# Reachable Stat. Pairs	Reach. Stat. Calc. Time (s)	# Iter. for fanin select.	Fanin select. Time (s)		
b01	8	0.37	9	0.10	1	0.47
b02	7	0.16	7	0.05	1	0.21
b03	2058	16	7	0.1	2	16
b04	N/A	>3h	N/A	N/A	1	>3h
b05	70	0.82	3	1.3	1	2.1
b06	7	0.37	6	0.1	1	0.46
b07	87	0.61	9	0.7	1	1.3
b08	29,186	54min or >3h	163	1min	1	55min or >3h
b09	N/A	>3h	N/A	N/A	1	>3h
b10	4,464	80	32	1	1	80
b11-b22	N/A	>3h	N/A	N/A	1	>3h

ターゲット回路の中で 2 信号で回路が修正出来るという解が得られたのは 9 個である。ターゲットを作成する前の回路のゲートはすべて 2 入力であるから、1 ターゲットでは 2 入力、2 ターゲットでは 4 入力の答えが出るのが自然である。しかし実験結果ではより少ない信号数で済む場合が多く見られた。特に 33 回路では入力信号数 0 となっており、ターゲットの値が 0 固定か 1 固定で構わないことになっている。これは、与えられた順序回路が冗長だったことを示唆している。一方で当初の信号数よりも多くの信号が必要となった場合もあった。これはセットカバリング問題で最適解が得られなかった場合や、到達可能状態組を選択する際に不必要な状態組が含まれてしまったことが原因として考えられる。

4.6.3　データパス FF 情報を用いた修正

タイムアウトしてしまった回路について追加の実験を行った。前述の通り、状態遷移に関わるフリップフロップはエンコーディングによって動作が異なるため対応を取るのが難しい場合がある。しかしデータパス系のフリップフロップの場合、2 回路の間で対応を取るのは簡単なはずであるし、すべての値を取りうると仮定すればプライマリ入出力と同等に考えることも出来る。追加の実験では、このようなデータパス系のフリップフロップの

表 4.6: Memory Usage (MB)

Name	Memory	Name	Memory
b01	31	b11	158
b02	31	b13	180
b03	104	b14	284
b04	257	b15	163
b05	56	b17	189
b06	32	b18	198
b07	94	b19	196
b08	159	b20	195
b09	182	b21	200
b10	49	b22	197

表 4.7: Number of patch inputs (Same Encoding)

# Target	# patch inputs					
	0	1	2	3	4	5+
1	15	8	11	2	2	0
2	5	9	10	4	2	6

表 4.8: Number of patch inputs (Different Encoding)

# Target	# patch inputs					
	0	1	2	3	4	5+
1	9	14	10	3	1	0
2	4	10	15	4	1	3

対応がすでに取れていて、あらゆる値を取りうるものとして実験を行った。残された状態遷移系のフリップフロップの数は表 4.3 に示されている。依然として状態遷移系のフリップフロップの関係性は分からないので、到達可能状態組を計算する必要はある。結果を表 4.9 に示す。ここでは、すべての回路について制限時間内（3 時間）にパッチ回路を生成することが出来た。なお、b17,b18,b19 は複数のモジュールに分かれており、この情報も

表 4.9: Buggy Circuits with Different Encoding, with Data-path FF Correspondence (average of 10 problems)

Name	In Final Loop				# Iter	Total ime (s)
	# Reachable Stat. Pairs	Reach. Stat. Calc. Time (s)	# Iter. for fanin select.	Fanin select. Time (s)		
b04	3	0.27	119	0.62	1	0.89
b08	4	0.24	35	0.16	1	0.40
b09	4	0.16	23	0.24	1	0.40
b11	9	0.30	87	0.7	1	1.0
b13	1,280	4.5	48	0.3	1	4.8
b14	8	8.0	553	1,165	1	1,173
b15	60	18	1,046	90	1	103
b17	180	180	1,313	1111	1	1300
b18	376	376	66	714	1	1090
b19	732	732	1	878	1	1610
b20	80	35	432	1,300	1	1,330
b21	80	37	844	2,880	1	2,945
b22	576	79	1,193	3,200	1	3,394

活用した点に注意したい。

既存の論理合成ツールを使用していればフリップフロップの対応は簡単に分かるはずだから、上述の仮定は納得できるはずである。しかし、そのような場合には合成前後の状態の割り当て情報も得られるはずであるから、非現実的だと考える読者がいるかもしれない。従って、さらに別の実験を行うために、各ベンチマーク回路に対して以下の 2 つの設計ファイルを用意した。この実験では、そもそも一方の状態遷移に誤りがあり、うまく状態の対応が取れていないことを想定している。

- VHDL ファイルを論理合成したもの（仕様）
- VHDL ファイルの状態遷移にバグを混入した上で論理合成し、その後ターゲットを設定した設計ファイル

1 つ目のファイルは仕様であり、もう 1 つが修正したい回路である。前述の通り状態遷移系のフリップフロップに当たる記述は容易に特定出来るので、バグを混入させるのも容易である。ターゲットには、バグが混入されたフリップフロップにつながるゲートを選択し

表 4.10: Buggy Circuits with Wrong State Transition, with Data-path FF Correspondence

Name	In Final Loop				# Iter	Total Time (s)
	# Reachable Stat. Pairs	Reach. Stat. Calc. Time (s)	# Iter. for fanin select.	Fanin select. Time (s)		
b04	3	0.18	22	0.35	1	0.54
b08	4	0.13	59	0.16	1	0.30
b09	4	0.12	17	0.12	1	0.27
b11	9	0.29	138	0.54	1	0.93
b13	1280	3.1	106	0.2	1	3.4
b14	8	10	428	1.3	2	23
b15	60	16	288	28	1	45
b17	180	180	1128	85	1	444
b18	376	376	4532	3497	1	3800
b19	732	732	1	792	1	1520
b20	80	36	81	98	1	143
b21	80	35	133	89	1	126
b22	576	65	2709	611	1	671

た。従って、本実験では必ず回路を修正できるようにターゲットを選んでいる。実験結果を表 4.10 に示す。提案手法は誤った状態遷移も正しく修正することが出来た。

以上のように、本節ではデータパス系 FF の対応情報を活用した実験を行った。表 4.4,4.5 にあるように、提案手法は 2-30,000 程度の状態組数を取り扱うのが限界である。しかし、データパス系フリップフロップの対応情報を用いることによって、全ベンチマーク回路も扱えた。一般に論理合成ツールはそのような情報をダンプするし、少なくともフリップフロップ名から対応が予想できることが多い。

本節の実験では、信号組の生成時間が支配的であり、到達可能状態組の計算時間は短かった。10,000 ゲート以上の回路だったことが原因である。

本節での実験はあらゆる順序回路の修正には適用できないことには注意されたい。異なるアルゴリズムで同じ関数を実現している 2 回路が与えられた場合、データパス系の FF の対応は取れない。似たアルゴリズムを採用していても、FF の値がちょうど否定の関係にある場合があり、この場合にも FF の対応は取れない。また、バグを含んだ順序回路が大きく誤っている場合、データパス系の FF の対応が取りにくくなる可能性もある。

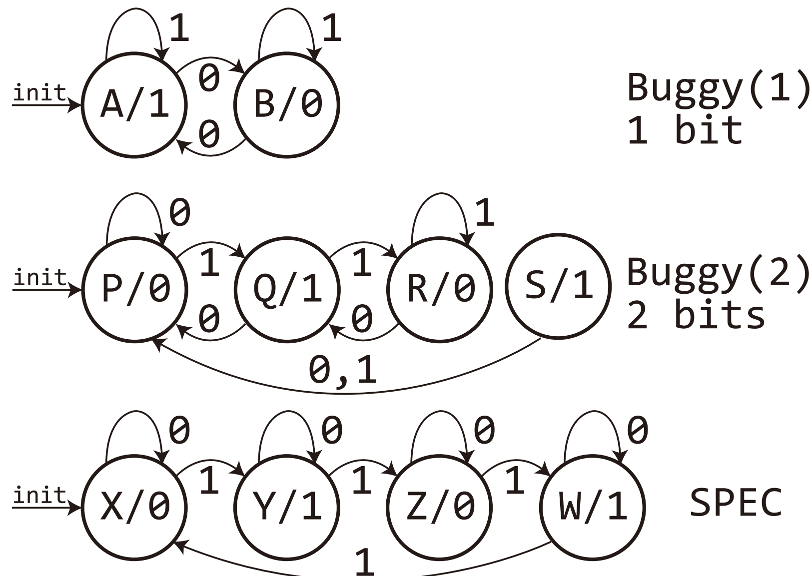


図 4.10: Experiment3: two buggy state machines and the specification

4.6.4　多対多の関係の状態組での実験

第 4.5 節の冒頭で、提案手法が 1 対 1・1 対多・多対多の状態関係に適用可能であることを述べた。これは、仕様回路と修正したい回路の 2 回路の状態組を用いて問題を解いているからである。しかし、これまでの実験では 2 回路はエンコーディングは違えど完全に同じ状態遷移を持ったものしかなかった。ベンチマーク回路と全く異なる状態遷移を持つような等価回路を予め用意するのは難しいからである。

そこで、本節では全く異なる 3 つのステートマシンを用意した。3 つを図 4.10 に示す。この実験では、図 4.10 の最後のステートマシンを仕様とし、残りの 2 つのステートマシンを修正することにする。3 つのステートマシンはゲートレベル記述に変換され、修正したい回路の中から 2 信号をターゲットとして設定した。3 つの状態遷移は明らかに異なるものである。

それぞれの回路は約 15 ゲートで構成され、フリップフロップ数は Buggy(1) で 1 個、それ以外では 2 個である。実験は修正した回路ごとに 10 回行われ、ターゲットがランダムに選ばれた。実行時間はすべて 1 秒以内程度であった。

ターゲットの選択によって解は異なった。Buggy(1) では 1 つの、Buggy(2) では 2 つの修正方法を見つけることが出来た。修正後の回路のステートマシンを図 4.11 に示す。もとの回路の状態"X,Y,Z,W"に対して、上から順に 1 対多、多対多、1 対 1 の関係性を見ることが出来、それらを図 4.12 に示した。以上のように、提案手法は 1 対 1 対応の回路だけでなく、1 対多や多対多の状態関係も取り扱うことが出来る。

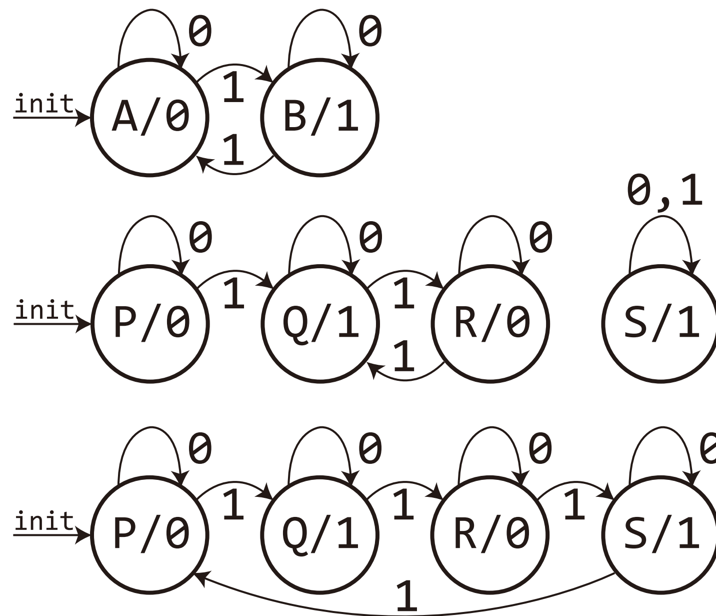


図 4.11: Experiment3: Results

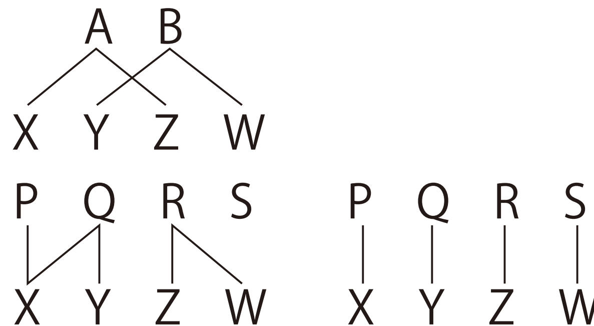


図 4.12: State correspondence: 1-1, 1-n, n-m

4.7　結論と今後の課題

4.7.1　結論

本章では、順序回路を自動修正するための信号選択手法を説明した。仕様回路と修正したい回路の間で到達可能な状態組をすべて計算する新手法を用いた上で、第 3 章の手法も活用した。実験では ITC99 ベンチマーク回路を用い、提案手法が正しいパッチ回路を生成できることを確認した。しかし、到達可能な状態組数が 30,000 程度が限度であり、それ以上は時間が掛かりすぎることも分かった。そこで、データパス系のフリップフロップの対応が取れていると仮定して、それらをプライマリー入力として扱って新たに実験を行った。すると探索すべき空間が大幅に削減され、ITC99 ベンチマーク回路のすべてを現

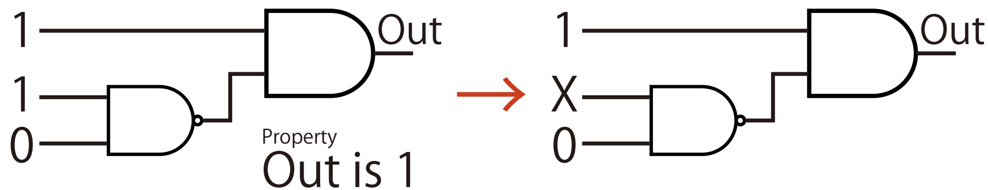


図 4.13: Minimizing Counter Example

実的な時間内で取り扱う事が出来た。また、手法は多対多の状態関係を取り扱う事ができ、そのことに関しても実験で確認した。

4.7.2　今後の課題

今後の方針は、より効率的な到達可能状態組の計算手法を考えることである。提案手法のような探索法は、深さ優先探索と考えることが出来る。深さ優先と幅優先探索で最終的な探索回数は変わらないが、再帰関数を用いた深すぎる探索ではメモリ使用量が膨大になってしまう。幅優先探索ならば省メモリで済む。また、修正のために考えるべきサイクル数が分かっている場合もあり、その際にも幅優先探索の方が実装がシンプルに済む。

[28] では、効率的な反例一般化手法が説明されている。ここでの一般化とは、探索中のプロパティを満たすような入力をより汎用的な形に変化させることである。図 4.13 のように、プロパティを満たすためには入力"110"の 2 ビット目の値は無関係である。このような場合に反例を"1-0"という形で得るための手法が説明されている。詳細は [28] を参照されたいが、このような一般化をおよそゲート数に比例する速さで行うことが出来る。

提案手法では、式 (4.3) と式 (4.5) の 2 つの SAT 問題が繰り返し解かれている。式 4.3 を以下に再掲する。これは状態組 (st_{spec}, st_{impl}) が Legal かどうかを調べる式で、Illegal の場合には SAT となる。

$$\begin{aligned}&\exists in.\ Spec(in, st_{spec}) \neq F(in, (0,0,\ldots,0), st_{impl})\\&\quad\wedge\ldots\\&\quad\wedge Spec(in, st_{spec}) \neq F(in, (1,1,\ldots,1), st_{impl})\end{aligned}$$

SAT の場合、命題を満たすような状態組 (st_{spec}, st_{impl}) は [28] の手法を用いて一般化することが可能である。従って、与えられた状態組 (st_{spec}, st_{impl}) が Legal かどうかを判定するだけでなく、Illegal だった場合には他の Illegal な状態組も見つけることが出来る。これによって探索が高速化する可能性がある。

式 (4.5) も以下に再掲する。この式は、与えられた状態組 (st_{spec}, st_{impl}) において、探

索されていない次状態を持つような入力 In を探すものである。

$$
\begin{aligned}
&\exists In. \\
&\quad Spec(In, st_{spec}) = F(In, (0, \ldots, 0), st_{impl}) \\
&\quad \rightarrow (Spec_{ST}(In, st_{spec}), F_{ST}(In, (0, \ldots, 0), st_{impl})) \notin R \\
&\quad \wedge \ldots \wedge \\
&\quad Spec(In, st_{spec}) = F(In, (1, \ldots, 1), st_{impl}) \\
&\quad \rightarrow (Spec_{ST}(In, st_{spec}), F_{ST}(In, (1, \ldots, 1), st_{impl})) \notin R
\end{aligned}
$$

上式を満たすような In が見つかった場合、[28] を用いてこの入力値を一般化することが出来る。入力 In の一部に"-(Don’t Care)"を与えると、出力にまで"-"が伝搬することがあり、調べるべき次状態が単一状態でなくなる。同時に複数の状態を調べることが出来るようになるから、探索が高速化する可能性がある。

注意したいのは、Algorithm4.3 を少し変更しなくてはいけない点である。16,22 行目では、Illegal な状態組や Illegal な次状態組しか持たない状態組を Unreach 配列に追加している。しかし、状態組が"-"を含んでいる場合には複数の状態組を表現しているので、実際に Unreach に追加されるべき状態組はその一部であることがある。そのことを直前の SAT 式の解などから確認しなくてはいけない。

以上のような拡張を行い、提案手法がより大きな回路にも適用できることを確認したい。

シンボリック探索

上述の拡張では依然として、巨大な回路では実行時間が長くなりすぎる可能性がある。Reach や Unreach には（"-"が含まれているとはいえ）個々の状態が積み重ねられており、莫大な状態組を扱うためには相応の回数の SAT 演算が必要となる。

Reach や Unreach の部分までシンボリックな表現（Characteristic function など）にできるような手法を導入できれば、10^{10} のオーダーの状態組数を取り扱うことが出来る。このような探索手法はシンボリック探索 [29] と呼ばれている。

第 5 章

修正されたゲートレベル回路からのC 記述再合成

5.1　既存研究

プログラム合成は古くから存在する研究分野である。人間が思い描く複雑なシステムを数千行のプログラムに落とし込むほどのことは出来ないので、様々な制約を与えた上で数ステートメントのプログラムを合成する方向性の研究が盛んである。

[30] では、様々なプログラム合成手法が調査されているが、以下の 3 点がそれぞれを特徴付けるとしている。

- 仕様の与え方
- 合成するプログラムの文法制約の与え方
- 合成の仕方

プログラム合成分野では SyGuS-COMP という大会が毎年開催されており、その大会では SyGuS [31] と呼ばれる記述で合成問題が出題されている。SyGuS は SMT ソルバで使われている SMT-LIB 記法に似ており、変数宣言・論理関係の記述・論理式での制約の記述が行える。SyGuS ソルバには様々なものが存在するが、SMT ソルバをベースとして独自の拡張を施しているものが多い。SyGuS-COMP で紹介されている Symbolic Solver [32] を例に、前述の 3 点との対応を示す。

- 仕様の与え方：論理式
- 合成するプログラムの制約の与え方：使用できる演算子・制御構文などを SyGuS 記法で制限
- 合成の仕方：CEGIS(詳細は後述、バックエンドとして SMT ソルバ)

最新の研究として [33, 34] が挙げられる。[33] ではより効率的に UNSAT の情報を活用する手法が提案されている。[34] では探索空間の削減に機械学習を用いる方法が提案されている。いずれの手法も、論理式として仕様を与え、SyGuS 記法で探索空間を制限し、CEGIS を用いる点に変わりは無い。

SyGuS と関連するプログラム自動合成手法として、Sketch [35] が挙げられる。ここではプログラムの一部分が空欄となったものに制約式を加えたものを合成器に与える。合成器では CEGIS(Counter Example Guided Inductive Synthesis) という手法が再現されており、効率的に合成が行えるようになっている。CEGIS 手法は本研究でも利用するため、詳細は後述する。以上を同様に 3 点に対応させる。

- 仕様の与え方：論理式
- 合成するプログラムの制約の与え方：一部が空欄になったプログラム
- 合成の仕方：CEGIS (詳細は後述、バックエンドとして SMT ソルバ)

以上既存研究を挙げたが、多くの既存研究は仕様として論理式が与えられるものがほとんどであった。仕様を自然言語で与える研究 [36] も存在するが、使用可能な表現が非常に限られているなど問題が多い。正確な論理式を書き下すことが出来る場合プログラムを書くことも難しくないとも考えられ、プログラム自動合成分野の研究を実際に活用するためには「どのように仕様を与えるか」という観点を大切にする必要があると筆者は考える。

5.2 目的

本章では C ベース設計において ECO が発生し、ゲートレベル記述は手動で修正することが出来た場合に、C 記述を自動修正することを考える。回路の設計者と C 記述の設計者が異なる場合にはこのような状況が発生する可能性がある。さらに、修正すべき C 記述がビット演算を含む場合、修正そのものが難しいので自動化の価値がある。

本章では、修正されたゲートレベル記述から、修正前の C 記述を自動修正する手法を提案する。ゲートレベル記述から C 記述全体を再生成することは難しくないが、もとの C 記述と全く異なり人間には理解の難しいものが得られる可能性がある [37]。従って、本章では修正前の C 記述を活用し、もとの C 記述と似た正しい記述を得ることとする。

修正済みのゲートレベル記述は仕様たりうるが、本章の研究では「ゲートレベル記述のシミュレータ」を仕様とする。仕様の与え方の難しさを前節で述べたが、本研究で考える「C 記述の自動修正」という例題では、自然な形で完全な仕様が与えられる点が特徴的である。またシミュレータを用いているので、大きな規模のゲートレベル記述でもシミュレータさえ高速に動けば良いことになる。

実験では、どの程度の大きさのプログラムに適用可能か調べるため、実験では AES256 ビット暗号化プログラムなども取り扱う。

5.3 問題の定義

以下の 3 つを入力とする。

- 仕様： ゲートレベル記述のシミュレータ
- テンプレート： 修正したい C 記述で、穴あきになっているもの
- 修正すべき場所で使用可能な変数・定数の数・演算子の種類・演算子数の情報

出力として穴あき箇所が埋められた正しいプログラムを得ることを目的とする。

入力によって必要サイクル数が変化する回路は取り扱いが難しいため、与えるゲートレベル記述は固定回数で出力が得られるもののみとする。テンプレートはいくつかの場所が穴あきになったプログラムであるが、この場所は設計者が事前に知っているものとする。

従って、設計者は ECO の内容を理解しており、C 記述の対応部分を見つけられる必要がある。小さな空欄でもあらゆるプログラム文が入ることになると、たった 1 行の空欄でも無限の探索空間を調べる必要が出てしまう。従って、空所内で使用可能な変数・定数の数・演算子の種類/数も与えられるものとする。修正後のコードがどのようなものになるか、ある程度の予想がついている必要がある。

空所部分に"if"や"while"などの制御文を含むことは出来ないので注意されたい。C 記述内に"for"や"while"などのループが含まれていても問題ないが、何回以内にループが抜けるかは簡単に分かり、事前に与えられるものとする。ポインタ演算を扱うにはメモリーモデルを構築する必要があり、探索空間が大きくなりすぎてしまうという問題点がある。実験では、ポインタ演算は行わないことにした。

5.4　提案手法

本節では、修正されたゲートレベル記述をもとに C 記述を自動修正するための提案手法を説明する。提案手法の大まかな流れは以下の通りである。また図 5.1 にも示す。

1. 修正内容を大まかに知っている開発者が、テンプレートを作成する。テンプレートは複数用意して良い
2. テンプレートとゲートレベル記述を入力として、修正済みの C 記述を自動生成する

1 つ目のテンプレートの作成方法については第 5.4.1 節で説明する。2 つ目の C 記述の自動生成技術については第 5.4.2 節で説明する。用意するテンプレートは 1 つでも複数でも良いが、解が得られない場合や誤った解が得られてしまった場合に備えて、複数あったほうが良い。後述する実験では 1 つしか与えない場合もある。

5.4.1　テンプレートの作成

本節ではテンプレートの作成方法を説明する。前述の通り、数千行のコードをすべて合成するのは困難である。従って提案手法では、元となるコードの一部分を穴あきとしたものを用いて合成を行う。コード 5.1 を元に合成することを考える。このコードは、入力"altitude（高度）"を元にプロペラの回転数"speed"を決める飛行船制御プログラムの骨格である。

プログラマはどのような修正を施すべきか、どの部分を変えれば良いか予想が付いていると仮定するので、以下のように空欄が設定できる。

さて、空欄内に入るのが int 型の整数であると仮定すれば、適切な数値を探索することが出来る。しかし、空欄にプログラム文が入ると考えると選択肢が無数に存在し絞り込む

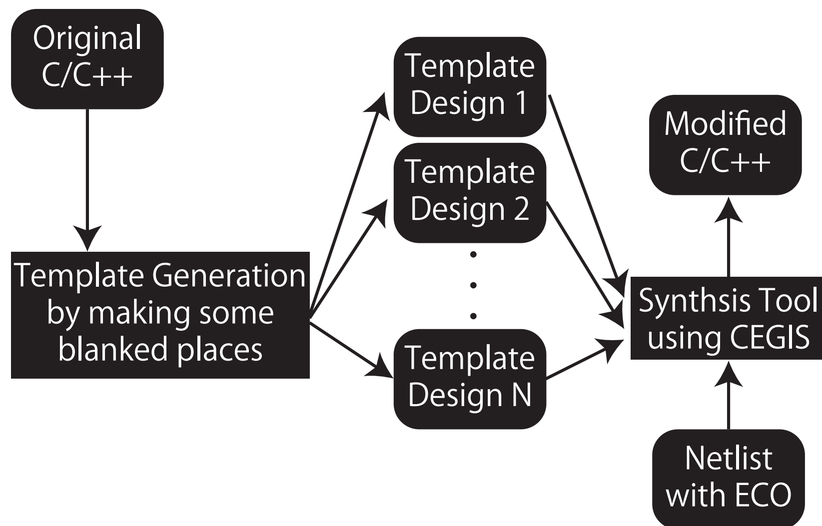


図 5.1: General flow for C Reconstruction

コード 5.1: Example (before making template)

```
1 ...
2 if(altitude>target){
3   speed=30;
4 }else if(lower<altitude){
5   speed=130;
6 }else{
7   speed=80;
8 } ...
```

ことは難しい。従って本手法では以下のような情報も追加で与える。なお、空欄中で関数を使うことはないものとする。

- どの演算子を使うか
- 演算子をいくつ使うか
- これまでに登場した変数の中で使用可能なものと、定数 (Integer 型) の数

この情報は探索空間を大幅に削減してくれるため、合成に当たって非常に重要である。しかしこれらの情報は人手で与える必要があり、修正内容を大まかに理解している必要がある点に注意したい。C 言語には算術・比較・論理・ビット演算子だけで 30 以上の種類があり、これを削減することは探索空間の削減につながる。演算子数も同様であり、同様に探索空間の大幅な削減につながるが詳細は後述の「ブロック」の紹介時に述べる。C 言語のプログラム文中で使用可能な変数はこれまでに登場したローカル変数かグローバル変数

コード 5.2: Example (making vacant place)

```
1   ...
2   if(altitude>target){
3     speed=30;
4   }else if(lower<altitude){
5     speed= /* Vacant Place 1 */ ;
6   }else{
7     speed=80;
8   } ...
```

コード 5.3: Example (making vacant place)

```
1   ...
2   if(altitude>target){
3     speed=30;
4   }else if(lower<altitude){
5 speed=
    /* "numop":3, "numconst":2,"var":["altitude"],"op":["+","-","*"] */ ;

6   }else{
7     speed=80;
8   } ...
```

に限られるが、行数の多い関数内の終盤に空欄が作成された場合には探索空間が膨大になってしまう可能性がある。また、使用可能な定数の数もその分だけ探索空間が増える。以上のように、探索空間が膨大になって実行時間が長すぎることのないように、予め以上の情報を与えている。

空所の中に情報を書き込んだものがコード 5.3 である。numop が演算子数、numconst が定数の数、var が使用可能な変数、op が使用可能な演算子の種類を表している。定数はすべて Integer 型とする。実装では、空所の中に JSON 形式で制約情報を与えて合成を行っている。

- "numop":3, "numconst":2, "var":["altitude"], "op":["+","-","*"]

JSON 形式のままでは合成出来ないので、等価な回路表現を用いて探索を行う。その基礎になる「ブロック」を図 5.2 に示す。これは MUX（マルチプレクサ）と ALU（演算回路）が組み合わされた構造になっており、図中の制御信号を調整することで様々なプログ

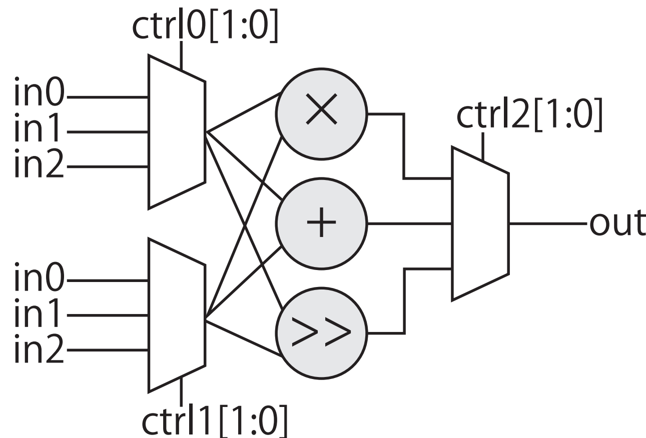


図 5.2: A Block

ラム文、例えば $in0 + in1, in1 >> in2, in2 \times in0$ などを表すことが出来る。 ctrl0,ctrl1 は使用する変数・定数などを選択し、ctrl2 は演算子を選択できる。ポイントは、プログラム文を表現するということを、ctrl 信号を選択するという問題に帰着させている点である。このことによって、無限に存在するプログラムの書き方を制限するとともに、最適なパラメータを探索することができる既存手法を活用してプログラム文を合成することが出来る。図中には示されていないが、未知の定数が存在する場合には、それも探索すべきパラメータに含めれば良い。

ブロック 1 つでは演算子 1 つのプログラム文しか合成出来ないから、それをつなげて複数の場合にも対応する。図 5.3 に示す。この図はコード 5.3 に合わせたものである。使用可能な演算子は加算・減算・除算の 3 つであり、 3 つの演算子が用意されている。一番左のブロックの入力として、2 つの定数 (Constant0, Constant1) と 1 つの変数 (altitude) が入力されている。注意したいのは、2 つ目のブロックでは 1 つ目のブロックの出力が、3 つ目のブロックでは 1 つ目と 2 つ目のブロックの出力が追加されている点である。これにより、$in0 + in1 \times in2$ や $(in0 >> in1) + in2$ などの複雑なプログラム文も生成出来るようになっている。

なお図 5.2 では、値の選び方が 3 通りの MUX が 3 つあるので、$3^3 = 27$ 通りのプログラムが考えられる。図 5.3 では 3 入力 MUX が 6 つ、4 入力 MUX が 2 つ、5 入力 MUX が 2 つあり、さらに定数値 2 つが穴となっている。定数が 8bit 変数であると考えると、$3^6\times\ 4^2\times\ 5^2\times\ (2^8)^2 = 19,110,297,600$ 通りのプログラムが考えられる。一般に、19,110,297,600 もの可能性を絞り込むほどの入出力組を用意することは 1 人のプログラマには不可能であると感じられる。しかし本手法を用いることで、高々 10 通りの入出力組に言及するだけで、これらの候補の中から正しいプログラムを得ることが出来る。なぜこのようなことが可能なのかは次節以降で述べる。

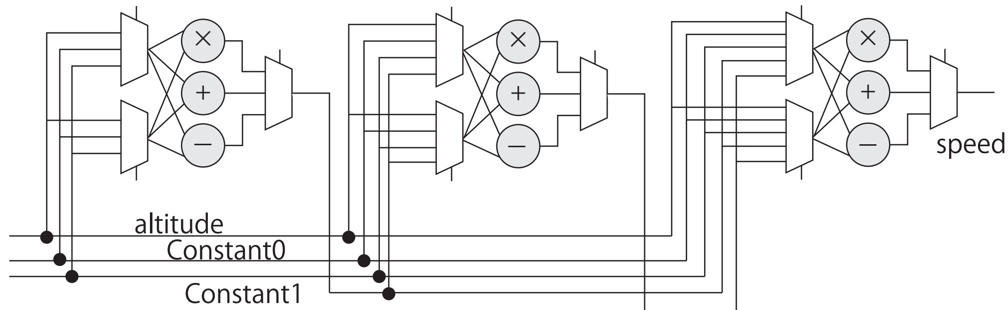


図 5.3: 3 Blocks

以上のように、C 記述の中に空欄を設けて文法的な制約を JSON 形式で記述することで、テンプレートが作成できることを示した。これによって、プログラム文の生成をパラメータ探索として考えることができ、機械的に探索することが可能になる。実装では、問題は SAT 問題に変換されて SAT ソルバで解けるようになっている。依然として探索空間は膨大であるように感じられるが、以降の手法や実験結果で手法がそれほど多くの入出力パターンを必要としないことを確認する。

5.4.2 プログラム自動部分合成手法の適用

5.4.2.1 一般的な定式化

本章では CEGIS(Counter Example Guided Inductive Synthesis) [35] について説明する。この手法を用いてプログラム文を合成することが出来る。

入力として、以下が用意されている。

- 修正済みのゲートレベル記述（仕様として利用）
- 元の C から作成したテンプレート

CEGIS のゴールは、これらの 2 つを入力として、テンプレート内のパラメータを決定してプログラム文を生成することである。仕様は $Spec$、テンプレートは $Tempalte$ と表記される。入出力を in, out とすると、$out = Spec(in)$ などと書き表せる。テンプレートにはパラメータ p が必要だから $out = Template(p, in)$ と表記する。以下の式 5.1 は、パラメータ p が満たすべき条件である。

$$\exists p \forall in \quad Template(p, in) = Spec(in) \tag{5.1}$$

すべての入力について Template の出力が Spec と等しくなる、という上述の条件は求めるパラメータそのものであるが、∀ を含む QBF 問題である。QBF 問題は SAT ソルバでは解けないので、第 3,4 章では ∀ の部分を展開して問題を解いていた。しかしこの場合、す

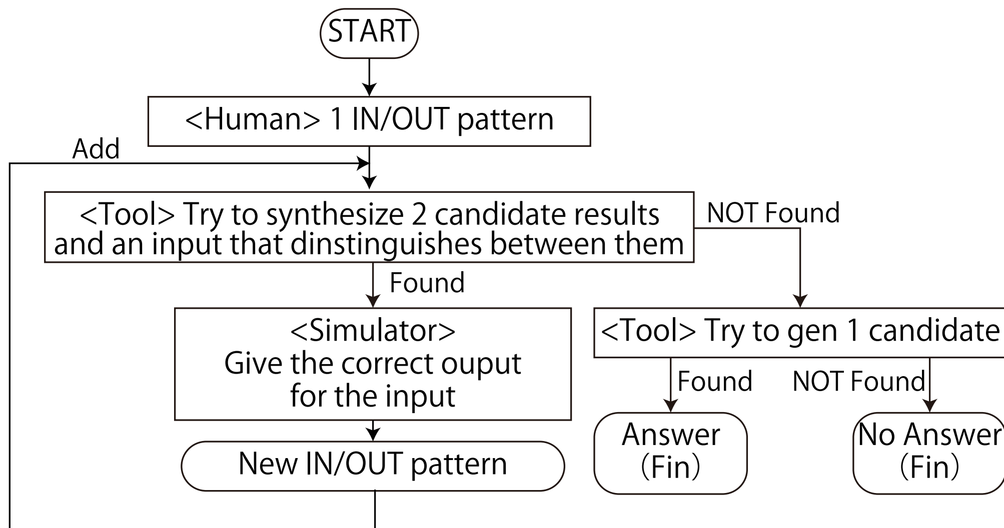


図 5.4: CEGIS flow

べての入力値について Template の出力と Spec の出力が等しくなることを確認せねばならず、現実的ではない。

そこで、CEGIS では入出力パターンを 1 つずつ増やしながらパラメータを探索することを考える。入出力パターンとは、仕様から得られる入力と正しい出力の組である。i 番目の入力値を IN_i とし、対応する出力は仕様のシミュレータを実行して得られるので $SIMULATOR(IN_i)$ と表すことにする。これ以降の数式では、= は全ビットが等しいことを、≠ は少なくとも 1 つのビットが異なることを表すものとする。おおまかな流れを図 5.4 に示す。

まずは、ある初期入力 IN_1 を用意してパラメータ探索を行う。以下の式 5.2 を解く。

$$\begin{aligned}
&\exists p_0, p_1, in \\
&\quad Template(p_0, IN_1) = SIMULATOR(IN_1) \\
&\quad \land Template(p_1, IN_1) = SIMULATOR(IN_1) \\
&\quad \land Template(p_0, in) \neq Template(p_1, in)
\end{aligned} \tag{5.2}$$

この数式では、2 つのパラメータの候補 p_0, p_1 を生成している点が特徴的である。さらに、これらのパラメータはある入力 in では異なる出力を持つようなパラメータである。もしこのような 2 パラメータが存在すれば、得られた in を新しい IN_2 と考えることが出

来る。すると、次は以下のような式 5.3 を解けば良い。

$$
\begin{aligned}
&\exists p_0, p_1, in \\
&\quad Template(p_0, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p_1, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p_0, IN_2) = SIMULATOR(IN_2) \\
&\quad \wedge Template(p_1, IN_2) = SIMULATOR(IN_2) \\
&\quad \wedge Template(p_0, in) \neq Template(p_1, in)
\end{aligned} \tag{5.3}
$$

以上のように in を繰り返し得ることを n-1 回繰り返すと、以下のような式 5.4 が得られる。

$$
\begin{aligned}
&\exists p_0, p_1, in \\
&\quad Template(p_0, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p_1, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p_0, IN_2) = SIMULATOR(IN_2) \\
&\quad \wedge Template(p_1, IN_2) = SIMULATOR(IN_2) \\
&\quad \wedge \ldots \\
&\quad \wedge Template(p_0, IN_n) = SIMULATOR(IN_n) \\
&\quad \wedge Template(p_1, IN_n) = SIMULATOR(IN_n) \\
&\quad \wedge Template(p_0, in) \neq Template(p_1, in)
\end{aligned} \tag{5.4}
$$

もしこの時点で式 5.4 が UNSAT となった場合、次のような 2 通りの場合が考えられる。

- 2 つのパラメータではなく、たった 1 つのパラメータが n 個の入出力パターンを満たす
- どのパラメータも入出力パターンを満たせない

以上のどちらかを調査するためには、以下の式 5.5 を解けば良い。

$$
\begin{aligned}
&\exists p. \\
&\quad Template(p, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p, IN_2) = SIMULATOR(IN_2) \\
&\quad \wedge \ldots \\
&\quad \wedge Template(p, IN_n) = SIMULATOR(IN_n)
\end{aligned} \tag{5.5}
$$

この式が SAT となった場合、これまでの入出力パターンをすべて満たすのは p だけとなり、プログラム合成の結果は p とすればよい。UNSAT となった場合には、仕様を満たすようなパラメータは存在しないことが分かる。注意したいのは、SAT となって得られた p が必ずしも正しい解とは限らない点である。p はあくまで n 個の入出力パターンしか調査

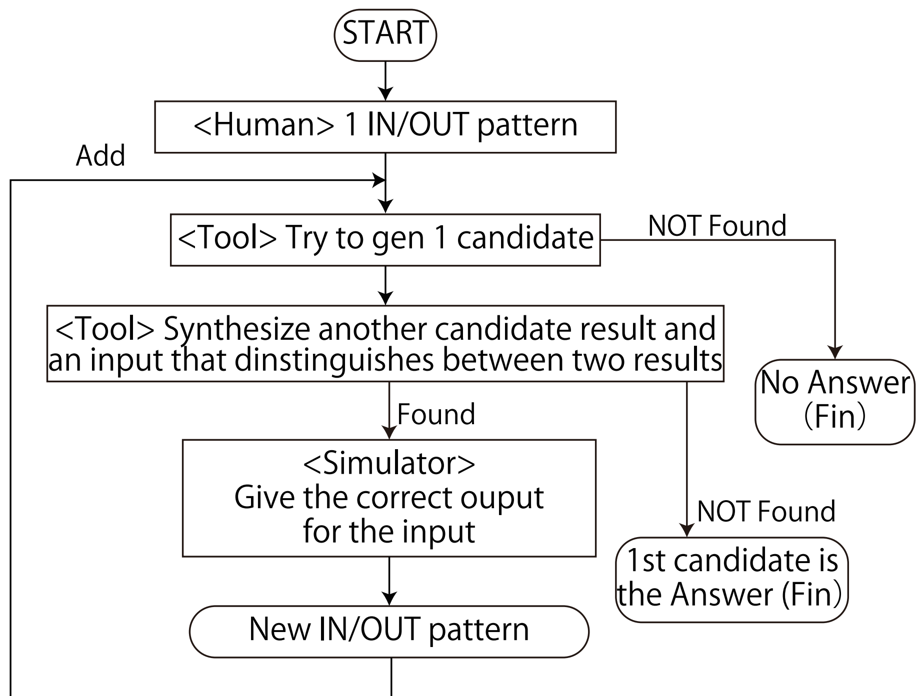


図 5.5: CEGIS flow for serial formulation

していないので、他の入力では正しい解が出せない可能性がある。このように、実際には解が存在しないテンプレートでも解が得られてしまうことがあることには注意したい。

この定式化の利点は、仕様が入出力組を得るための Simulator としてだけ用いられている点である。SAT 問題を作成するに当たって仕様の論理式は必要ないので、仕様が巨大な場合や論理式にしにくい場合にも適用可能である。

上式では n 個の入出力パターンを想定しているが、実際にはどの程度が必要だろうか。もし n の値が大きすぎると SAT ソルバに与える問題が大きくなりすぎ、現実的な時間内に解き終わらない可能性がある。実験では、せいぜい 10 個程度の入出力組があればパラメータを得られることが示された。ただしこのことに数学的な裏付けはないことには注意したい。

5.4.2.2 Serial な定式化

前節で CEGIS の一般的な定式化を説明したが、本節では異なるアプローチを説明する。おおまかな流れを図 5.5 に示す。図 5.4 との大きな違いは、パラメータを 2 つ一度に求めていない点である。以下で詳細を説明する。

n 個の入出力パターンが得られているとき、Serial な定式化ではまず以下の式 5.6 を

解く。

$$
\begin{aligned}
&\exists p. \\
&\quad Template(p, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p, IN_2) = SIMULATOR(IN_1) \\
&\quad \wedge \ldots \\
&\quad \wedge Template(p, IN_n) = SIMULATOR(IN_n)
\end{aligned}
\tag{5.6}
$$

上式が UNSAT となれば、与えられたテンプレートの範囲内で仕様を満たすパラメータは存在しないことになる。SAT となった場合、これ以上別の入出力パターンが必要かどうかも確認するために、以下の式 5.7 を解く。ただし、式 5.6 で得られたパラメータは P とする。

$$
\begin{aligned}
&\exists p, in. \\
&\quad Template(p, IN_1) = SIMULATOR(IN_1) \\
&\quad \wedge Template(p, IN_2) = SIMULATOR(IN_1) \\
&\quad \wedge \ldots \\
&\quad \wedge Template(p, IN_n) = SIMULATOR(IN_n) \\
&\quad \wedge Template(p, in) \neq Tempalte(P, in)
\end{aligned}
\tag{5.7}
$$

上式が SAT となって新しい p が得られれば、前節でパラメータが 2 つ得られた場合と同じである。このときは、同時に得られた in が次の IN_{n+1} となる。UNSAT となった場合、n 個の入出力パターンを満たすパラメータは P だけとなるので、解を P とすれば良い。

この Serial な定式化を用いる利点は、式のサイズが小さくなる点である。式 5.4 では $2n$ 個の Template 回路を用意しなければならなかったが、Serial では半分になる。その代わりに解く回数が 2 倍になってしまっている。実験結果では、2 倍の大きさの問題を一度に解くよりも、半分の大きさの問題を 2 度解く方が高速であるとの結果が得られている。

5.5　実験

本節では上述の提案手法の実験結果を説明する。提案手法は [38] を拡張する形で実装された。CBMC (C Bounded Model Checker) はオープンソースの有界モデル検査ツールであり、C 言語などで書かれたプログラムのプロパティが満たされるかどうかを検査することが出来る。CBMC はプログラムを等価な論理式に変換し、ソルバを用いて問題を解いている。CNF の他に SMT 形式に対応するが、本実験では CNF 式を生成し、バックエンドソルバとして MiniSAT [6] を用いた。テンプレートは CBMC に解きやすい形に自動で変換された。実験は以下の環境でおこなわれた。プログラムはすべてシングルスレッド実行である。

コード 5.4: Bitcount

```
uint32_t bit_count(uint32_t x){
  uint32_t num=0, mask=1, i;
  for(i=0; i<32; i++){
    if( mask&x )
      num++;
    mask=mask<<1;
  }
  return num;
}
```

- CPU: Core i7-3770 (3.4GHz)
- Memory: 16GB
- OS: Linux Kernel 4.10

5.5.1　ビットカウント

5.5.1.1　実験内容

1 つ目の例題として、ビットカウントを用いる。ビットカウントとは、32bit の入力に対して 1 の数を数えて出力するものである。以下が入出力例である。

- INPUT: b10000000111000000000000000000101
  -> OUTPUT:6
- INPUT: b11111111111111111001000101011111
  -> OUTPUT:25

もとの C 記述はコード 5.4 に示す。このプログラムは、for 文を使って下位 1bit ずつ 1 の数を数えるアルゴリズムである。

上述のソフトウェアはわかりやすいが、制御文を含まない高速なアルゴリズムも提案されている。実装を図 5.6 に示す。本実験では、図 5.6 が回路として実装されており、以下のような仕様変更が発生したとする。それぞれの仕様変更に対応した回路は既に得られているものとする。このような回路変更は、入力の一部をビット反転させたりすれば良いので回路上ではあまり難しくない。

- ECO1: 1 ではなく 0 の数を数える
- ECO2: 下位 16 ビットだけ数える
- ECO3: 下位 16 ビットの 1 の数と、上位 16 ビットの 0 の数を数える

コード 5.5: Bitcount Template 1

```
1  uint32_t bit_count(uint32_t x){
2    uint32_t num=0, mask=1, i;
3    for(i=0; i<[a]; i++){
4      if(  [b]  )
5        num++;
6      mask=mask<<1;
7    }
8    num =  [c] ;
9    return num;
10 }
```

コード 5.6: Bitcount Template 2

```
1  uint32_t bit_count(uint32_t x){
2    uint32_t num=0, mask=1, i;
3    for(i=0; i<[a]; i++){
4      if(  [b]  )
5        num++;
6      mask=mask<<1;
7    }
8    for(i=0; i<[d]; i++){
9      if(  [e]  )
10       num++;
11     mask=mask<<1;
12   }
13   num =  [c] ;
14   return num;
15 }
```

さて本実験の目的は、各仕様変更に対応する C 記述を得ることである。本実験では 2 つのテンプレート（コード 5.5,5.6）を用意した。筆者が人手で探索したところ、コード 5.5 は ECO3 に対応するには不十分であり、コード 5.6 が必要である。

それぞれの空欄は以下のような制約のもとで探索する。

- 空所 a,d は 32 ビットの定数である。JSON 記述は以下の通り。
  `{"numop":0, "numconst":1, "var":[], "op":[]}`

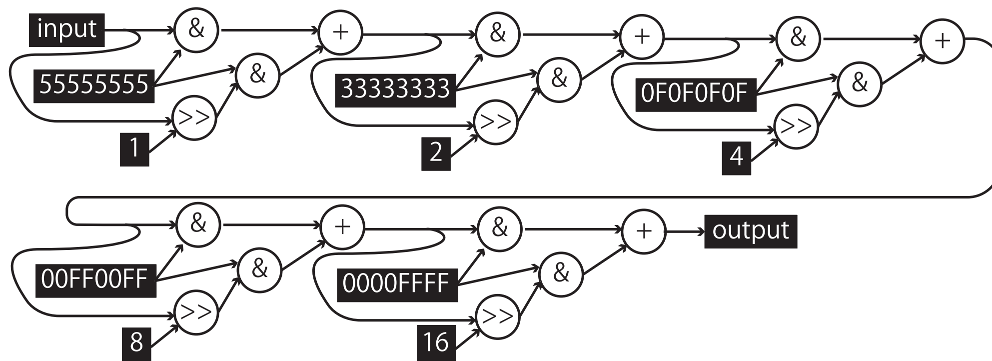


図 5.6: Fast algorithm for bitcount

- 空所 b,e はプログラム文であり、in,mask,1 定数が使用可能である。使用可能な演算子は 1 つであり、"&","|","»"に限る。JSON 記述は以下の通り。
  `{"numop":1, "numconst":1, "var":["in", "mask"], "op":["&","|","»"]}`
- 空所 c はプログラム文であり、num,1 定数が使用可能である。使用可能な演算子は 2 つであり、"&","+","-"に限る。JSON 記述は以下の通り。
  `{"numop":2, "numconst":1, "var":["num"], "op":["&","+","-"]}`

5.5.1.2 実験結果：2 つの定式化の違い

結果を表 5.1,5.2 に示す。 1 つ目の表は第 5.4.2.2 節で紹介した定式化を用いており、 2 つ目の表は第 5.4.2.1 で示した定式化を用いている。

ECO1 の場合、[c] に当てはまる正しい解は"32-num"である。0 の数を数えたいなら、1 の数を数えてから引き算すれば良い。Serial な定式化の場合、"16+16-num"が得られており、これは正しい。CEGIS の元の定式化を用いた場合、“2147483664-num+2147483664”が得られている。これも実際には"32-num"と同様であり、正しい。

ECO2 の場合、[c] にあてはまる正しい解は"num"である。Serial な定式化の場合、“num-2197815699+2197815699”が得られているが、これは"num"と等しい。元の定式化を用いた場合、“(num&num)|(num&num)”となっており、これも少し複雑になってしまっているが正しい。注意したいのは、"num"と同じ表現は"num-1+1","num-100+100"などたくさんある点である。提案手法の中では。これらの解は形すら違えど同じものと見なされる。

ECO3 はコード 5.5 では不十分であり、実際に Serial な定式化では答えが得られなかった。一方で元の定式化では 4 つの入力パターン (0,2168728634, 128, 16384) の時に解が得られてしまっている。しかし、コード 5.5 はテンプレートとして不十分で解を持たないことが Serial な定式化で示されているので、表 5.2 の答えは誤りであることが分かる。全探索したところ、この誤った解は全入力パターン 2^{32} 個のなかで約 2.5% の場合にしか正し

表 5.1: BitCount (Serial Formulation)

#	[a]	[b]	[c]	[d]	[e]	Time	# Pat.	Largest SAT
ECO1	39	in&mask	16+16-num	N/A	N/A	442s	6	(583090, 1575197)
ECO2	16	in&mask	num-2197815699 +2197815699	N/A	N/A	501s	7	(656890, 1790274)
ECO3 (Code 5.5)			No Solution	N/A	N/A	209s	5	(418633, 1129597)
ECO3 (Code 5.6)	16	in&mask	num+4294967275	37	in&mask	203m	7	(1324409, 3627190)

表 5.2: BitCount (Original Formulation)

#	[a]	[b]	[c]	[d]	[e]	Time	# Pat.	Largest SAT
ECO1	36	in&mask	2147483664-num+2147483664	N/A	N/A	635s	6	(1140643, 3133115)
ECO2	16	in&mask	(num&num)|(num&num)	N/A	N/A	1446s	10	(1787523, 4905387)
ECO3 (Code 5.5)	39	in&mask	(8|num)+8	N/A	N/A	286s	4	(817203, 2246979)
ECO3 (Code 5.6)	16	in&mask	num-9	25	in&mask	376m	8	(2938274, 8126100)

くならないことが分かった。従って、ランダムないくつかの入力パターンで解を確認するだけで誤りを見つけられる可能性がある。

コード 5.6 は ECO3 を表現出来ることは筆者が確認済みであり、2 つの定式化の両方で解を得ることが出来た。表 5.1 で、"num+4294967275"は"num-21"と等価である。[d] はループの回数を決めているが、16 回で十分なところを 21 回多く実行するプログラムとなっている。従って、num の値を 21 減少させることで辻褄が合うようになっている。このように、正しい解にもいくつかの候補があるが、実験結果ではそのうちの 1 つが得られている。

すべての実験で、ループの実行回数、つまり入出力パターンの数は 10 以下であった。Serial な定式化はもとの定式化よりもおよそ 2 倍の速さで問題が解けていることも確認できた。

表 5.3: Using different initial use-case (BitCount, Serial Formulation, ECO1).

Initial Pattern	time	# Patterns	Largest SAT problem
(0,32)	442 sec	6	(583090, 1575197)
(1,31)	453 sec	5	(448202, 1131904)
(983055, 24)	342 sec	4	(448264, 1132338)

表 5.4: Using different initial use-case (BitCount, Original Formulation, ECO1).

Initial Pattern	time	# Patterns	Largest SAT problem
(0,32)	635 sec	6	(1140643, 3133115)
(1,31)	1098 sec	6	(1204707, 3133115)
(983055, 24)	2707 sec	7	(1302331, 3576183)

表 5.5: AES implementation

# Inputs	# Outputs	# Latches	# Gates
72	165	7354	89085

5.5.1.3　実験結果：初期入出力組の影響

本節では、異なる初期入出力を与えた時に実行時間やループ回数がどのように変化するかを確認する。実験結果を表 5.3,5.4 に示す。この実験では ECO1 だけ行っている。

初期入出力の 1 つは (0,32) であり、前節でも用いられたものである。他に (1,31),(983055,24) も用いた。必要な入出力組数は初期入出力によって異なることがわかり、実行時間も変化した。ある初期入出力組が少ない入出力組数しか要求しないことはあるが、だからといって実行時間が短くなるとは限らず、むしろ長くなることもある。その意味で、良い入出力とは何か、どのように得るかを考えることは難しい。

異なる入出力組の場合でも、Serial な定式化が元の定式化よりも実行時間が短い傾向に変化はなかった。また、同じ入出力でも定式化によって異なる入出力組数が必要な場合もあることが確認できた。

コード 5.7: AES Template 2

```
1  char sbox[256]={0x63,0x7c,0x77,0x7b
2 //char sbox[256]={[ ]a], [ ]b], [ ]c], 0x7b,...
3 // First three values are synthesized.
```

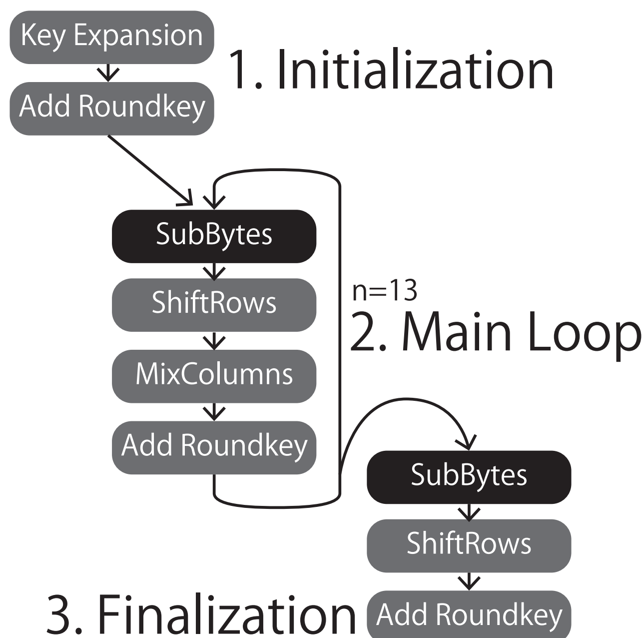


図 5.7: AES calculation

5.5.2　AES

本節では、提案手法のスケーラビリティを確認する。提案手法では仕様はシミュレータとして動作すれば良いから、大きなものでも取り扱う事が出来る。従って本節では、どれほど大きなテンプレートを扱えるかに着目する。

例題として、AES256 ビット暗号化プログラムを用意した。このコードは 200 行程度の C 記述であり、9,000 ほどの実行ステップ数を持つ。AES256 ビット暗号化では、図 5.7 にあるように、同じ内容の演算が 13 回繰り返されている。このコードをもとにテンプレートを作成する。テンプレートでは、コード 5.7 にあるように、SBOX 配列の始め 3 つの定数を空欄とした。本実験の目的は、空欄となった 8 ビット幅の定数の値 3 つを生成出来るか確認することである。なお SBOX 配列は図 5.7 の中の"SubBytes"関数で参照される。

仕様として、AES256 ビット暗号化を行うデジタル回路のネットリストを用意した。このネットリストの情報を表 5.5 に示す。この実装を何度かシミュレーションすることで、入力に対する正しい出力を得ることが出来る。

表 5.6: AES (Serial formulation))

	Get What?	Time	Vars	Clauses	Result	Memory
1	One param.	96min	278,214	1,273,173	SAT	596MB
2	Another param.	644min	851,410	5,021,260	SAT	2178MB
3	One param.	1.2min	556,530	1,860,028	SAT	496MB
4	Another param.	1157min	1,121,630	5,417,814	SAT	3077MB
5	One param.	4.3min	834,814	3,165,753	SAT	784MB
6	Another param.	0.1min	N/A	N/A	UNSAT	73MB
Total		1903min				3077MB

表 5.7: AES (Original formulation))

	Get What?	Time	Vars	Clauses	Result	Memory
1	Two params	>60 hours	1,137,395	6,326,414	N/A	
Total		>60 hours				

Serial な定式化を用いた場合の実験結果を表 5.6 に示す。32 時間ほどで正しい解を得ることが出来た。"Vars","Clauses"はそれぞれ SAT 問題の変数数・節数である。最も大きな SAT 問題の規模が 100 万変数、500 万節であるが、この 100 万変数は今日の SAT ソルバが現実的な時間内に問題を解くための限界であると言われている。従って、用意した AES のテンプレート (9,000 実行ステップ程度) は提案手法が適用可能な規模のおよそ限界であると考えられる。

元の定式化を用いた場合の実験結果は表 5.7 に示す。ここでは、最初の SAT 問題が 60 時間以内に解き終わらず、解を得ることが出来なかった。SAT 問題の規模は表 5.6 の中のどの SAT 問題よりも大きく、問題そのものも Serial な定式化よりは複雑だと思われる。前節と同様、本実験でも Serial な定式化は元の定式化よりも高速に解を得ることが出来た。

5.6　結論

本章では、仕様となるゲートレベル記述から C 記述を自動修正する手法について説明した。C 記述を一から生成するのではなく、元の C 記述の一部を変えたテンプレートを用いた。テンプレート内の空欄には文法的な制約を開発者が与えることで、探索空間を削減

することとしている。CEGIS 手法を用いて解が探索され、実験では正しい C 記述が生成出来ていることを確認した。仕様はシミュレータ上でしか用いられないから、提案手法の実行時間はテンプレートの大きさに応じて変化する。9,000 実行ステップ程度の AES 暗号化プログラムのテンプレートが 32 時間ほどで解き終わることを確認した。

提案手法の問題点は、解のないテンプレートを与えられた際に誤った解を得る可能性がある点である。この点については、いくつかのテンプレートを用意したり、ランダムな入出力を使って得られた C 記述の正しさを確認するなどの対策が考えられる。

実験結果では、人間が素直に開発すれば"32-num"とすべきところが、"16+16-num"などと合成される事があった。テンプレートに必要以上の自由度がある場合、分かりにくい C 記述が得られる可能性がある。いくつかの候補を示して、最も人間に分かりやすいものを選択させたりする工夫が考えられる。実験で得られた分かりにくいプログラム文にはいくつかのパターンがあった。

- １つの演算子で表される計算が、複数の演算子に分割されて分かりにくい場合 ($16 + 16 - num$ など)
- オーバーフロー・アンダーフローが発生している場合 ($num + 2197815699$ など)

1 つ目の場合は、演算子数を少なくして再合成すれば分かりやすいプログラム文に出来る可能性がある。2 つ目の場合は、Integer 型定数の範囲を制限して合成してみることが考えられる。

テンプレートの作り方には改善の余地がある。本研究では修正すべき場所・修正内容に大まかな当たりを付ける必要があった。どの内部変数を使うべきかを自動探索する手法と組み合わせるなど、よりシンプルなテンプレート作成方法が求められる。

第 6 章

結論と今後の課題

6.1　結論

本稿では LSI 論理設計の自動デバッグ・ECO 手法として、以下の 2 つの状況を想定した。

- 仕様と異なるゲートレベル記述を直接直したい場合（回路の自動部分修正）
- C ベース設計で ECO が発生し、ゲートレベル記述は直せたが、C 記述を自動で直したい場合（C 記述の自動部分修正）

6.1.1　回路の自動部分修正

1 つめの状況に対応するための提案手法として、第 3 章では修正のための信号選択手法を提案した。手法は複数修正すべき場所（ターゲット）がある場合に利用可能なものである。複数の修正すべき場所を同時に考えながら、修正のための内部信号を探索する定式化を提案した。この定式化を用いることで、ターゲットを 1 つずつ扱う既存手法よりも品質の良い解が得られる。ICCAD’17 コンテストで用いられたベンチマーク回路では、既存手法よりも最大 100 倍評価指標の良い解が得られることが確認できた。

ターゲットを 1 つずつ扱う既存手法は解の品質が良くないが、一方で大きな回路でも適用可能であることも確認でき、依然として有用である。そこで、既存手法で解けなかった回路を解析し、ターゲット順が重要であることも発見した。回路の構造から良いターゲット順を生成する手法も提案した。

第 4 章では組合せ回路だけでなく順序回路にも適用可能な信号選択手法を提案した。特に、第 3 章で提案した組み合わせ回路のための手法を適用するために、仕様回路と修正したい回路の 2 状態の中で到達可能なものだけを網羅する手法を提案した。実験では ITC99 ベンチマーク回路を用い、2 時間で 3 万状態組程度まで取り扱える事を確認した。また、データパス系 FF が分かっているという前提の元でならば ITC99 ベンチマーク回路すべてが取り扱えることを確認した。

6.1.2　C 記述の自動部分修正

2 つめの状況に対応するための提案手法として、第 5 章で C 記述の自動修正手法を提案した。修正済みの正しいゲートレベル記述をすでに得られていると仮定し、修正前の誤った C 記述を自動修正することが目的である。提案手法は、修正したい C 記述のどの部分を直せば良いかや、どの変数・演算子を使って直せば良いかは設計者が知っていると仮定し、プログラム自動合成手法を用いて正しい C 記述を得る。合成すべきプログラム文は複数のマルチプレクサで表現され、その制御信号の値（パラメータ）を探索することでプ

ログラム文を合成することを提案した。正しいパラメータを得るには、いくつかの入力パターンに対して正しい出力が得られるパラメータを探索し、複数のパラメータ候補がある場合には、新たに 1 つの入出力パターンを追加して再探索する。パラメータ探索は、候補が 1 つだけになるまで繰り返す。修正済みのゲートレベル記述はシミュレーターとしてだけ利用されるので、提案手法はゲートレベル記述の規模に関わらず利用できる。AES256 ビット暗号化プログラムなどを例題として用い、数千ステップの演算であれば自動部分合成できることを示した。

以上のように、本稿では自動デバッグ・ECO のなかでも特に困難と思われる状況を想定し、それぞれについて新手法を提案した。手法はどの程度の大きさの回路・プログラムを取り扱えるかという観点で評価されている。

6.2　今後の課題

6.2.1　回路の自動部分修正

第 3 章の信号選択手法では、各配線に重みが割り当てられており、その合計重みが少なくなるような信号組を探索した。ICCAD'17 CAD コンテストベンチマークをすべて現実的な時間内で解くことができており、スケーラビリティには大きな問題はないと考える。しかし、その他の部分では改善の余地がある。実際の開発現場では、パッチ回路の大きさも重要だと考えられ、修正に必要な信号の重みだけでは不十分だと思われる。どのようにすればゲート数の少ないパッチ回路が得られるか、という観点での研究は今後の課題としたい。また、実際に得られた解に基づいて再配置配線をし直してはいないので、より現実に即した指標を使ったり、配置配線ツールとの融合が考えられる。

第 4 章の順序回路での信号選択手法は、直したい回路と仕様回路の間の状態の対応付けを 1 つずつ調べていくものであった。提案手法だけでは適用可能な回路規模に限界があり、実験結果では最大 30,000 程度の対応付けの網羅が限界であった。この問題点を解決することが、最も重要な問題である。

まず 1 つに、より大きな規模の回路が扱えるよう、複数の状態付けを同時に扱えるような探索手法が求められる。[28] では PDR [12] で用いられている反例の一般化手法が説明されており、ゲート数に比例する程度の速さで反例を一般化することが出来る。ここでの一般化とは、例えば"1110"という反例を"1-1-"という形にすることである。この例では、"1110"だけでなく、"1011"なども反例となることが分かる。第 4 章の最後では手法の拡張の方向性を述べたので、この実験を行いたい。しかし、$10^{1}0$ オーダーの状態組数を持つような回路では依然として実行時間がかかりすぎる可能性がある。手法をシンボリック探索 [29] に拡張することなどが求められる。

また、実験では大きな回路を扱うためにデータパス系のフリップフロップの対応情報を用いた。仕様回路と修正したい回路の間ではデータパス系 FF の対応が取れているものとし、入出力と同等に扱うことで探索空間を大幅に削減することに成功した。この対応関係を自動で推測する手法が確立できれば、これまでの本提案手法の適用の幅を広げることが出来る。

6.2.2　C 記述の自動部分修正

第 5 章の C 記述の自動部分修正では、AES などの大規模な例題にも適用可能なことが確認されており、スケーラビリティの点では大きな問題はないと考える。一方で、テンプレートの作り方が難しいという指摘が考えられる。テンプレートは元の C 記述の一部分を空欄にし、その空欄部分に以下の情報を与える。これらをすべて用意するのはプログラマの負担になる。

- 使用可能な変数
- 定数の数
- 演算子の種類・数

第 3,4 章の信号選択手法のように「事前に使用可能な変数を調べる」ということをするためには、仕様が論理式の形で与えられている必要がある。本研究では高速化のために仕様はシミュレーターとしたが、前提を変えることで信号選択手法の応用は可能であると考える。仕様が得られていれば、得られた解が本当に正しいかどうか等価性検証を行うことも出来るようになる。

シミュレーターを使って解を得た場合には、その解が本当に正しいかどうか検証できないという問題点がある。いくつかのランダムな入出力パターンを用いてシミュレーターの結果と比較することが考えられる。誤った解がたまたま見つかってしまう確率がどの程度なのかについても調査の余地がある。

得られるプログラム文が必要以上に複雑で、人間に分かりにくいという問題もある。例えば、"num+16+16"というプログラム文は、"num+32"とした方が分かりやすい。この原因には、演算子数が多すぎる場合や、オーバーフロー・アンダーフローが使われた場合がある。これらを回避するには、演算子数を少なくして再生成したり、合成される定数に制限を設ける事などが考えられる。複数の候補を挙げて、分かりやすいものを人間に選ばせても良い。

提案手法ではマルチプレクサを数珠つなぎにしたものを用いてプログラム文を自動生成したが、他の与え方も考えられる。[39] は、X-Btree と呼ばれる構造を用いて様々な論理を表現することを提案している。表現がコンパクトになるように工夫がされており、プロ

グラム文の自動部分合成にも活用可能であると考えられる。

謝辞

この場を借りて、本研究を行うにあたってお世話になった皆様にお礼を申し上げたいと思います。指導教員の藤田昌宏教授には学部 4 年生からの 6 年間、研究に関して様々な形でご指導頂きました。国内外から沢山の研究者が訪問する研究室に身を置くことができたお陰で、様々な研究テーマに触れることができ、アカデミックな雰囲気の中で自由に研究に取り組むことが出来ました。論文執筆にあたっては度々たくさんの助言を頂くことが出来ました。本当にありがとうございました。

坂井修一先生、中山雅哉先生、田浦健次朗先生、佐藤周行先生には、本論文の副査として様々なご指摘をいただきました。ありがとうございました。

Amir Masoud Gharehbaghi 助教には、様々な実験をする上での具体的なアドバイスを頂きました。論文誌の執筆に当たっては内容面から英語面まで沢山の助力を頂きました。ありがとうございました。

松本高士助教には、研究面だけでなく VDEC のツールやネットワークに関する様々なアドバイスを頂きました。ありがとうございました。

藤田研秘書の長澤しのぶさんには、様々な書類手続きでお世話になりました。ありがとうございました。他にも研究室内の先輩・後輩に助けていただきながら、研究を進めることが出来ました。ありがとうございました。

最後に、研究生活を全面的に支援してくれた両親に感謝したいです。2 人の生活面・精神面・金銭面でのサポートなしには、この論文は決して書き上がらなかったと思います。本当にありがとうございました。

参考文献

[1] H. Foster. 2018 fpga functional verification trends. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pp. 40–45, Dec 2018.

[2] A. M. Gharehbaghi and M. Fujita. A new approach for debugging logic circuits without explicitly debugging their functionality. In *2016 IEEE 25th Asian Test Symposium (ATS)*, pp. 31–36, Nov 2016.

[3] Thomas R. Nicely. Pentium fdiv flaw. `www.trnicely.net`.

[4] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, May 2019.

[5] H. Foster. Trends in functional verification: a 2016 industry study. In *DVCon 2017*, 2017.

[6] Niklas Een and Niklas Sorensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pp. 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[7] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, 2013.

[8] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, Vol. 7, No. 3, pp. 201–215, July 1960.

[9] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, Vol. 5, No. 7, pp. 394–397, July 1962.

[10] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD ’96, pp. 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Satoshi Jo, Takeshi Matsumoto, and Masahiro Fujita. Sat-based automatic rectification and debugging of combinational circuits with lut insertions. *IPSJ Transactions on System LSI Design Methodology*, Vol. 7, pp. 46–55, 2014.

[12] Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '07, pp. 173–180, Washington, DC, USA, 2007. IEEE Computer Society.

[13] M. Fujita, T. Kakuda, and Y. Matsunaga. Redesign and automatic error correction of combinational circuits. *Logic and Architecture Synthesis*, pp. 253–262, 1991.

[14] Daniel Brand, Anthony Drumm, Sandip Kundu, and Prakash Narain. Incremental synthesis. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, pp. 14–18, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[15] S. L. Huang, W. H. Lin, P. K. Huang, and C. Y. Huang. Match and replace: A functional eco engine for multierror circuit rectification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 32, No. 3, pp. 467–478, March 2013.

[16] Y. S. Yang, S. Sinha, A. Veneris, and R. K. Brayton. Automating logic transformations with approximate spfds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 30, No. 5, pp. 651–664, May 2011.

[17] Ai Quoc Dao, Nian-Ze Lee, Li-Cheng Chen, Mark Po-Hung Lin, Jie-Hong R. Jiang, Alan Mishchenko, and Robert Brayton. Efficient computation of eco patch functions. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pp. 51:1–51:6, New York, NY, USA, 2018. ACM.

[18] A. M. Gharehbaghi and M. Fujita. A new approach for selecting inputs of logic functions during debug. In *2017 18th International Symposium on Quality Electronic Design (ISQED)*, pp. 166–173, March 2017.

[19] Y. Kimura, A. M. Gharehbaghi, and M. Fujita. Signal selection methods for efficient multi-target correction. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2019.

[20] N. Lee, V. N. Kravets, and J. R. Jiang. Sequential engineering change order under retiming and resynthesis. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 109–116, Nov 2017.

[21] C. Huang, C. Hsu, C. Wu, and K. Khoo. Iccad-2017 cad contest in resource-aware patch generation. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 857–862, Nov 2017.

[22] Mikoláš Janota and Joao Marques-Silva. Abstraction-based algorithm for 2qbf. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pp. 230–244, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] Valeriy Balabanov, Jie-Hong Roland Jiang, Christoph Scholl, Alan Mishchenko, and Robert K. Brayton. 2qbf: Challenges and solutions. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pp. 453–469. Springer International Publishing, 2016.

[24] L. H. Goldstein and E. L. Thigpen. Scoap: Sandia controllability/observability analysis program. In *17th DAC*, pp. 190–196, June 1980.

[25] IBM. Cplex. `www.ibm.com/products/ilog-cplex-optimization-studio`.

[26] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 26, No. 9, pp. 1124–1137, Sept 2004.

[27] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification. `www.eecs.berkeley.edu/~alanmi/abc`.

[28] Alan Mishchenko, Niklas Een, and Robert Brayton. A toolbox for counter-example analysis and optimization. IWLS '13, 2013.

[29] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pp. 317–320, June 1999.

[30] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pp. 13–24, New York, NY, USA, 2010. ACM.

[31] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, Oct 2013.

[32] Sygus Competition Group. Symbolic solver. `https://github.com/rishabhs/sygus-comp14`.

[33] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pp. 420–435, New York, NY, USA, 2018. ACM.

[34] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *SIGPLAN Not.*, Vol. 53, No. 4, pp. 436–449, June 2018.

[35] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, Vol. 15, No. 5, pp. 475–495, Oct 2013.

[36] S. Balkovski and I. G. Harris. Designing cyber-physical systems from natural language descriptions. In *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 39–44, Oct 2017.

[37] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik. Reverse engineering digital circuits using structural and functional analyses. *IEEE Transactions on Emerging Topics in Computing*, Vol. 2, No. 01, pp. 63–80, jan 2014.

[38] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[39] Hiroaki Yoshida and Masahiro Fujita. Exact minimum factoring of incompletely specified logic functions via quantified boolean satisfiability. *IPSJ Transactions on System LSI Design Methodology*, Vol. 4, pp. 70–79, 2011.

発表文献

Journals

- （５章）○ Y. Kimura, A. M. Gharehbaghi, M. Fujita, “C Description Reconstruction Method from a Revised Netlist for ECO Support”, IEICE transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2018
- （４章）○ Y. Kimura, A. M. Gharehbaghi, M. Fujita, “Signal Selection Methods for Debugging Gate-Level Sequential Circuits”, IEICE transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2019
- (準備中, ４章) ○ Y. Kimura, et-al, "Debugging Large Gate-level Sequential Circuits"

Conference Papers

- ○ Y. Kimura, M. Fujita, "Specification by existing design plus use-cases," 2016 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2016
- Q. Wang, ○ Y. Kimura, M. Fujita, "Automatically adjusting system level designs after RTL/gate-level ECO," 2016 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2017
- M. Fujita, ○ Y. Kimura, Q. Wang, "Template based synthesis for high performance computing," 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), 2017
- Qinhao Wang, ○ Y. Kimura, M. Fujita, "Methods of equivalence checking and ECO support under C-based design through reproduction of C descriptions from implementation designs," 2017 18th International Symposium on Quality Electronic Design (ISQED), 2017
- ○ Y. Kimura, A. M. Gharehbaghi, M. Fujita, “Patch Function Input Selection Methods for Efficient Multi-Target Rectification“, 2018 International Workshop on Logic

Synthesis, 2018
- （３章）○ Y. Kimura, A. M. Gharehbaghi, M. Fujita, "Signal Selection Methods for Efficient Multi-Target Correction," 2019 IEEE International Symposium on Circuits and Systems (ISCAS), 2019
- Masahiro Fujita, ○ Yusuke Kimura, Xingming Le, Yukio MIyasaka, Amir Masoud Gharehbaghi, "Synthesis and Optimization of Multiple Portions of Circuits for ECO based on Set-Covering and QBF Formulations", 2020 Design Automation and Test in Europe Conferenc (Date), 2020

国内会議・研究会

- ○木村 悠介, ガラバギ アミル マスード, 藤田 昌宏, 「実時間組み込みソフトウェア解析のための HW/SW 協調検査」, 研究報告システムと LSI の設計技術（SLDM）, 2015
- ○木村悠介, 石山薫太郎, 藤田　昌宏, 「有界モデル検査ツールを用いた C 言語プログラム部分合成」 , 情報処理学会 DA シンポジウム 2016
- 王勤浩, ○木村 悠介, ガラバギ アミル マスード, 藤田 昌宏, 「RTL 設計時の ECO のためのテンプレートを用いた C 記述合成手法」, 研究報告システムと LSI の設計技術（SLDM）, 2017

表彰

- ○木村悠介, 「情報処理学会 SLDM 研究会　優秀学生発表賞」, 2015
- ○ Yusuke Kimura, Peikun Wang, Yukio Miyasaka, Kentaro Iwata, Xingming Le, Xiaoran Han, “ICCAD 2017 CAD Contest, 3rd prize”, 2017