競技プログラミングコンテストの提出プログラムを利用した記号実行ツール KLEE の定量的評価

岡野 兼也　吉田 則裕　山本 椋太　高田 広章

ソフトウェアの大規模化・複雑化により人手での網羅的なテストケース作成が困難になってきている．このことから，網羅的なテストケース自動生成には期待が寄せられている．テストケース自動生成を目的とした記号実行ツール KLEE が開発されている．約 160 のソフトウェアに KLEE を適用したところ，生成されたテストケースのコードカバレッジは 90%を超えたと主張されている．しかし，対象としたソフトウェアのほとんどは，システムソフトウェアと呼ばれるものであり，多様性に欠ける．そこで，多様なアルゴリズムが利用されている，競技プログラミングコンテストの提出プログラムに KLEE を適用し，定量的な評価を実施した．評価の結果，シンボルとなる変数が利用するメモリ使用量の大きい場合に KLEE が異常終了しやすく，プログラムの循環的複雑度は KLEE の生成するテストケースのコードカバレッジに影響を与えないということがわかった．

The increases in the scale and complexity of a software system make the manual creation of exhaustive test cases more difficult. The automatic generation of exhaustive test cases is needed to alleviate it. A symbolic execution tool KLEE has developed for the automatic test case generation. It is insisted that the code coverage is greater than 90% when KLEE is applied to approximately 160 targets. However, most of the targets are categorized into system software and lack variety. Therefore, we performed a quantitative evaluation of KLEE with competitive programming archives that include the implementations of various algorithms. As a result, we found that KLEE tends to terminate abnormally when symbol variables consume much memory. Also, the cyclomatic complexity of a target program makes an insignificant impact on the code coverage of the generated test cases.

1　はじめに

ソフトウェアの信頼性や機密性を確かめるためには，十分なソフトウェアテストの実施が必要不可欠である．しかし，ソフトウェアの大規模・複雑化により，ソフトウェアテストは非常に時間のかかる作業であり，開発工程のうち 4 割弱を占める [2]．この割合は年々増加しており，大きな問題となっている [8]．

以上から，ソフトウェアテストの自動化には高い期待が寄せられている．自動化可能なテストケースの生成手法の 1 つに，記号実行を利用したものがある．記号実行はプログラムを擬似的に実行し，実行可能な経路を網羅的に抽出する技術である [5]．記号実行の実行中にでは変数をシンボルとして扱い，1 つの実行経路を取りうる変数の値を導出することができる．この特性を利用すると，論理的にはプログラムのすべての実行経路を網羅したテストケースを生成することが可能である．記号実行の研究は 1970 年代から学術的には盛んに行われ，多くの成果を挙げている．しかし，記号実行の計算コストの高さなどに起因して，実際のソフトウェア開発の現場で利用されることはほとんど無かった．2000 年代になると，計算機の性能向上や，充足可能性判定問題を高速に解く SAT (satisfiability problem) Solver の飛躍的進化 [3] から，十分現実的な時間で記号実行によるプログラムの解析が可能になった．

記号実行を利用したテストケース生成ツールとして最も有名なものに，KLEE [1] がある．KLEE は LLVM (Low Level Virtual Machine) ビットコード

Quantitative Evaluation of Symbolic Execution Tool KLEE with Competitive Programming Archives.
Kenya Okano, 株式会社サイバーエージェント, CyberAgent, Inc.
Norihiro Yoshida, Ryota Yamamoto, Hiroaki Takada, 名古屋大学, Nagoya University.

に変換された C 言語のプログラムに対して記号実行を行い，テストケースを作成するツールである．実際に Coreutils や Busybox を始めとする，約 150 の OSS (Open Source Software) に対して適用されている．生成されたテストケースは平均で 90%のコードカバレッジを実現し，脆弱性を発見することにも成功している．また，商用の静的解析ツールと比較した評価 [6] も行われており，静的解析ツールでは検出することができなかったソフトウェアの脆弱性の発見に成功している [1]．しかし，対象としたソフトウェアのほとんどはシステムソフトウェアと呼ばれるものであり，多様性に欠ける．

そこで，本研究では多様な特性を持つプログラムを利用して，KLEE を定量的に評価することを目的とする．目的を達成するべく，競技プログラミングコンテスト運営サイト AtCoder †1 に提出されたプログラムに対して KLEE を適用した．競技プログラミングコンテストでは，数学的問題を解決するために多くのアルゴリズムやデータ構造が利用される．この特徴を生かし，多様な特徴を持つプログラムに KLEE を適用した結果と，プログラムから得られるメトリクスの関係を調査する．また，どのようなプログラムであれば KLEE を利用したテストケースの自動生成に向いているのか，あるいは向いていないのかを考察する．

2 関連研究

2.1 記号実行ツール KLEE

記号実行はプログラムを擬似的に実行し，実行可能な経路を網羅的に抽出する技術である [5]．実行中には変数にシンボルを割り当てることで，具体値を伴わない値として扱う．記号実行では，条件分岐によって分かれるすべての経路を網羅的に実行する．1 つの実行経路における分岐条件は，実行終了時にまとめられ，そのすべての条件を組み合わせたものを経路制約と呼ぶ．経路制約を SAT Solver に与えることで，実行経路に対応した変数の具体値を求めることができる．求められた具体値は，1 つ実行経路での挙動を確かめるためのテストケースに対応する．

記号実行を利用したテストケース生成ツールとして最も有名なものに，KLEE [1] がある．LLVM ビットコードに変換された C 言語のプログラムに対して記号実行を行い，テストケースを生成する．同時に，実行可能な経路の総数や，実行経路に対応した変数の具対値を出力する．

前述したように記号実行には計算コストが非常に大きいという欠点が存在する．この問題点を解決するために KLEE では SAT Solver に与える経路制約を事前に簡単化する，キャッシュを利用して複数回の問い合わせがある場合に高速化を図るなどの工夫が行われている．

2.2 KLEE の評価

KLEE は Open Source Software（以下 OSS）を利用して，性能の評価がされている [1]．対象には Coreutils に含まれる 89 のプログラム，BusyBox に含まれる 75 のプログラム等が利用されている．

はじめに，Coreutils を対象とした評価について紹介する．対象とした Coreutils ではそれぞれ開発者の手によってテストプログラムが公開されている．そこで，KLEE の実行により生成されたテストケースと，開発されてきたテストプログラムでコードカバレッジの比較がなされた．KLEE の実行により生成されたテストと，開発者が人手により作成したテストプログラムのコードカバレッジ分布を表 1 に示す．さらに，15 の Coreutils のプログラムに対しては，ランダムテストを加えてコードカバレッジを比較している．KLEE の実行により生成されたテストと，開発者が人手により作成したテストプログラムとランダムテストでのコードカバレッジの差を図 1 に示す．表 1，図 1 を確認すると，KLEE を利用することでコードのカバレッジが大きく上昇している．

次に，BusyBox を対象とした評価について紹介する．BusyBox についても，開発者によって作成されたテストプログラムとの比較がなされている．KLEE の実行により生成されたテストと，開発者が人手により作成したテストプログラムのコードカバレッジ分布を表 2 に示す．BusyBox でも同様に，KLEE を利用して生成したテストケースではコードカバレッジが高

†1 https://atcoder.jp/

表 1　Coreutils を対象としたコードカバレッジの分布

カバレッジ	KLEE	開発者
100%	16	1
90%-100%	40	6
80%-90%	21	20
70%-80%	7	23
60%-70%	5	15
50%-60%	-	10
40%-50%	-	6
30%-40%	-	3
20%-30%	-	1
10%-20%	-	3
0%-10%	-	1
テスト対象全体のカバレッジ	84.5%	67.7%
アプリごとのカバレッジの中央値	94.7%	72.5%
アプリごとのカバレッジの平均値	90.9%	68.4%

表 2　Busybox を対象としたコードカバレッジの分布

カバレッジ	KLEE	開発者
100%	31	4
90%-100%	24	3
80%-90%	10	15
70%-80%	5	6
60%-70%	2	7
50%-60%	-	4
40%-50%	-	-
30%-40%	-	2
20%-30%	-	1
10%-20%	-	-
0%-10%	-	30
テスト対象全体のカバレッジ	90.5%	44.8%
アプリごとのカバレッジの中央値	97.5%	58.9%
アプリごとのカバレッジの平均値	93.5%	43.7%

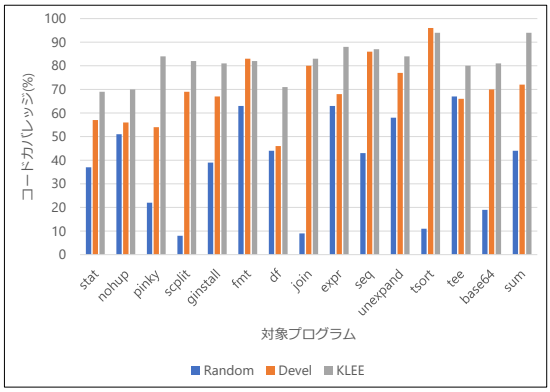

図 1　ランダムテストと人手で作成されたテストと **KLEE** で生成されたテストの比較 [1]

くなっている．

論文のまとめとして，Coreutils や Busybox など合計 150 以上のソフトウェアで，平均 90%以上のコードカバレッジのテストケースを生成できたため，KLEE を利用したテストケースの自動生成は十分に実用的であるという評価がされている．

以上から，KLEE はテストケースの生成や網羅的な解析において大きな成果を上げている．しかし，調査対象となっているプログラムはシステムプログラムと呼ばれるものばかりである．よって，プログラムの特徴には偏りがあるということが予想され，異なる特徴を持つプログラムを利用して評価がされていない．そのため，多様なプログラムによるテストによってその性能を確認する必要がある．

3　評価実験

本研究では以下のリサーチクエスチョン (**RQ**) に基づいて，KLEE を評価する．

RQ1　KLEE をプログラムに適用した時に，KLEE の実行時エラーが発生しないかどうか．

RQ2　10 分間で 10 個以上の実行経路に対するテストケースが生成されるかどうか．

RQ3　生成されたテストケースのコードカバレッジが 50%を超えるかどうか．

まず，これらの項目をなぜ **RQ** としたのか解説する．**RQ1** は，KLEE を異なる多くの特徴を持つプログラムに対して適用した時に，正しく実行が完了するかどうかを調べるための項目である．ソフトウェア

テストを自動化するためには，できるだけ多くのプログラムで KLEE が正常に動作することが好ましい．あるいは，KLEE の得手不得手を明らかにして，別のツールと組み合わせる必要がある．そこで，KLEE がどのような特徴を持つプログラムに対して正常に動作するのかを調べる必要がある．**RQ2** は，KLEE によるテストケースの生成が現実的な時間で可能かどうかを調べるための項目である．SAT Solver の性能が向上しているが，記号実行が持つ分岐に対して指数的に計算量が増加するという問題は，本質的には解決していない．そこで，一定の時間で KLEE の実行を打ち切った際に，どの程度テストケースが生成されるのかを調べる必要がある．**RQ3** は，KLEE により生成されたテストケースがどの程度プログラムを網羅的に実行するのか調べるための項目である．ソフトウェアテストを自動化するにあたり，KLEE で生成されるテストケースは実行経路を網羅的に実行できる必要がある．そこで，KLEE により生成されたテストコードが，どの程度のコードカバレッジとなるのか調べる必要がある．

以降，本研究で行った 2 つの実験について記述する．

3.1 競技プログラミング

競技プログラミングとは，参加者に問題が与えられ，より早く与えられた問題の条件を満たすプログラムを記述する競技である [7]．

多くの競技プログラミングの問題には，問題文，入力形式とその制約，出力，メモリ制約，時間制約が記述されている．問題文は処理の内容，入力形式とその制約はどのような入力となるか，出力はどのような出力にする必要があるか，メモリ制約はプログラムの実行メモリの最大値，時間制約はプログラムの実行時間の最大値を示している．一般に入力は標準入力，出力は標準出力を利用して行う．問題にはメモリ，時間制約が存在するため，競技者は問題を解くための空間計算量，時間計算量を意識してプログラムを記述する必要がある．そのため，問題の難易度が上昇すると素朴なアルゴリズムを利用した解答では制約を満たすことが困難になり，様々なデータ構造，応用的なアルゴリズムが求められる．また，これらの制約が存在するために，競技プログラミングでは処理速度が高速で，実行時のメモリ使用量が少ない C 言語/C++が好んで利用される．

多くの競技プログラミングコンテストでは，問題を解くために利用したプログラムをオンラインジャッジシステムに提出する．オンラインジャッジシステムとは，ユーザから提出されたプログラムを必要に応じてコンパイル，厳格なテストデータを入力として実行することで，提出プログラムが問題に即しているか検査するシステムである [7]．多くのオンラインジャッジシステムでは，検査結果はリアルタイムで競技者にフィードバックされ，プログラムが受理された際にはコンテストの得点に反映される．

以上から，競技プログラミングコンテストの提出プログラムは以下のような特性を持つことがわかる．

- 問題で指定された条件を満たすため，多くのデータ構造やアルゴリズムを利用する．
- C 言語/C++等のプログラミング言語が利用される．
- オンラインジャッジシステムで受理されたプログラムはすべて，問題用に設定されたテストケースに対して正常に動作する．
- プログラムは標準入力から与えられ，決められた入力によってのみ振る舞いを変える．

また，これらの特徴は多様な特徴を持つプログラムに対して KLEE を適用するという目的に即している．

3.2 実験方法

まず，本節では実験で共通している部分について説明を行う．実験ごとに異なる部分については 3.3, 3.4 節でそれぞれ述べる．

3.2.1 実験の流れ

実験方法の概要を以下に記述する．以降，各項目での詳細を説明する．

1. 競技プログラミングの提出プログラムを取得
2. 入力変数を KLEE のシンボルに変更
3. Clang で LLVM ビットコードに変換
4. KLEE の実行
5. KLEE の実行結果や，プログラムのメトリクスを収集

6. 評価

まず，競技プログラミングの提出プログラムを競技プログラミングコンテスト運営サイト AtCoder から収集した．アルゴリズムの偏りを避けるため同一問題のプログラムは 10 以下となるようにした．

次に KLEE を適用するための準備を行った．KLEE を適用するためにはシンボルをどの変数に割り当てるのか，シンボル値の制限にはどのようなものがあるのかをプログラムに記述する必要がある．3.1 項で説明したように競技プログラミングコンテストの提出プログラムは，標準入力によってのみ振る舞いを変える．そこで，標準入力を受け取っている変数をシンボルとし，標準入力から入力を受け付けるプログラムをコメントアウトすることで KLEE の適用を可能にした．なお，シンボル値の制約については実験によって変更しているため実験ごとに説明する．ただし，不要な経路探索を行わないようにし，高速化を図るため AtCoder の入力制約の範囲内でシンボル値の制限を行っている．

プログラムの変更後，Clang を利用してプログラムを LLVM ビットコードに変換した．このとき，インクルードパスに KLEE 用のヘッダファイルのあるディレクトリを指定し，KLEE のシンボルを制御するための関数群を利用できるようにした．

ここまでで KLEE の実行準備が整ったため，作成した LLVM ビットコードに対して KLEE を適用した．実行環境や KLEE に与えたオプション等については 3.2.2 に記述する．

KLEE の実行後，評価するためのメトリクスを収集した．プログラムのメトリクス収集には SourceMonitor †2 を利用した．SourceMonitor はプログラムの関数の数，行数，循環的複雑度，ブロックのネスト数などの静的メトリクスを測定するためのツールである．C++, Java, C#など様々なプログラミング言語に対応しており，プロジェクト全体の測定結果，ファイルごとの測定結果など用途に合わせて出力することが可能である．C 言語のプログラムに対して SourceMonitor で取得可能なメトリクスを表 3 に示す．なお，これらのメトリクスは改行が適切になされない場合や，マクロが利用されている場合等には正常に計測できないものがある．

表 3　SourceMonitor で取得したメトリクス

メトリクスの名称	説明
Lines	プログラムの行数
Statements	命令数
% Branches	制御構文の数
% Comments	コメントの割合
Functions	関数数
Avg Stmts/Function	命令数の平均/関数数
Max Complexity	循環的複雑度の最大値
Max Depth	関数内の最大ネスト数
Avg Depth	関数内の平均ネスト数
Avg Complexity	循環的複雑度の平均

KLEE の実行結果については KLEE の出力ファイルや KLEE の補助コマンドの ‘‘`klee-stats`’’ の実行結果を利用して情報を収集した．また，コードカバレッジについては ‘‘`klee-stats`’’ コマンドのうち “ICov (%) ”，“BCov (%) ” の項目を利用して調べた．これらはそれぞれ，命令カバレッジ，ブランチカバレッジを示す．

さらに，`time` コマンドを利用してプログラムの実行メモリも測定した．測定には AtCoder で提供されているオンラインジャッジのテストケースを利用し，すべてのテストケースを入力した際の最大のメモリ使用量をプログラムの実行メモリとして扱った．

最後に，収集したメトリクスを利用して定義した **RQ** に対してどのような結果になったか確認することで評価を行った．

3.2.2　実験環境

本実験で KLEE を実行した環境について表 4-6 に示す．

本実験を行うにあたりオペレーティングシステムに行った調整として，プロセスのスタックサイズの上限変更がある．これは，記号実行を行った時にシンボルのサイズが大きく，まとめられた経路制約が複雑なものになった場合，SAT Solver が再帰を繰り返すため

†2　http://www.campwoodsw.com/sourcemonitor.html

表 4　オペレーティングシステムの環境

仮想化マシン	VMware Workstation Player 15
CPU	CPU: Core i5-7300U @ 2.60GHz
メモリ	Memory: 6GiB
OS	Ubuntu 16.04 LTS
プロセスのスタックサイズの上限	無制限

表 5　コンテナの環境

Docker バージョン	18.09.0
コンテナイメージ	klee/klee:latest (2018/12/17)
メモリ	4GiB
プロセスのスタックサイズの上限	無制限

表 6　KLEE の実行制約

打ち切り時間	10 分
メモリ使用上限	3GiB
フォーク数上限	15000

である．制限を緩和することで SAT Solver は KLEE に割り当てられているメモリに余裕がある限り，再帰呼出しが可能となる．

また，KLEE の実行環境には公開されている Docker Image[†3] を利用した．理由として，インストールの方法や依存パッケージの推奨バージョンの違いなど細かい環境による差をなるべく少なくすることが挙げられる．

3.2.3　結果の分類

実験を行ったところ，結果は以下のように大別されることがわかった．以降，実験結果を示す際に下記の表記を利用する．

GREEN

KLEE の実行時エラーが起こらず，10 以上の実行経路で対応したテストケースが生成された．あるいは，KLEE の実行時エラーが起こらず，KLEE の経路探索により発見された実行経路のうち，半分以上で実行経路に対応したテストケースが生成された．なお，KLEE で実行を途中で打ち切った場合経路探索やテストケースの具体値の導出が完了していない場合がある．

SLOW

KLEE の実行時エラーが起こらず，10 未満の実行経路に対応したテストケースが生成される．かつ，KLEE の実行時エラーが起こらず，経路探索により発見された実行経路のうち，半分未満の実行経路に対応したテストケースが生成される．

SIGSEGV

セグメンテーションフォールトが発生し，KLEE が異常終了する．

SHAREDMEM

“LLVM ERROR: not enough shared memory for counterexample ” というエラーが発生し，KLEE が異常終了する．

FORKERR

“ERROR: fork failed (for STP) ” というエラーが発生し，KLEE が異常終了する．

3.3　競技プログラミングの制約に合わせた実験

3.3.1　評価対象

本節の実験では競技プログラミングの問題で与えられた入力値の制約をそのまま KLEE のシンボル制約として利用する．

3.3.2　結果

プログラムに 3.2.1 項に記述した手順で KLEE を適用したところ 100 のプログラムのうち 18 が **GREEN**，9 が **SLOW**，30 が **SIGSEGV**，38 が

†3　https://hub.docker.com/r/klee/klee/

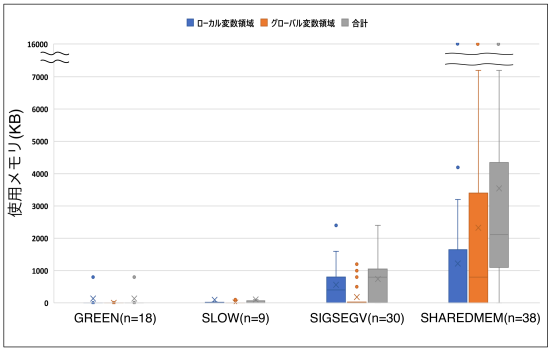

図 2　3.3 でシンボルとした変数のメモリサイズ

SHAREDMEM となった．残りの 5 プログラムについては実行ごとに結果が大きく変化するため，定義しなかった．結果ごとのメトリクスの平均値を表 7 に示す．なお，本実験では正常に KLEE の実行が終了したプログラムが 27/100 と非常に少数であったため，コードカバレッジについては評価していない．

また，KLEE の実行中にユーザがシンボルとして指定した変数に割り当てるメモリサイズが大きく，低速になるかクラッシュするという旨の警告がほとんどのプログラムで発生した．そこで，シンボルとした変数に割り当てたメモリの大きさを調査した．その結果を図 2 に示す．

図 2 を見ると，異常終了しているプログラムでは正常終了しているプログラムに比べて明らかにシンボルとした変数に割り当てられているメモリサイズが大きい．

行った実験から，シンボルとした変数のメモリサイズが大きい場合に記号実行に影響を及ぼし，KLEE の実行が失敗しているということが推測できる．そこで，3.4 節ではシンボルとなる変数に与えるメモリサイズを小さくし，再度プログラムに KLEE を適用した．

3.4　入力サイズを制限した実験

3.4.1　評価対象

3.3 節の実験で，シンボルとした変数に与えるメモリのサイズが大きいと KLEE は異常終了することが多いということがわかった．そこで，本節の実験では

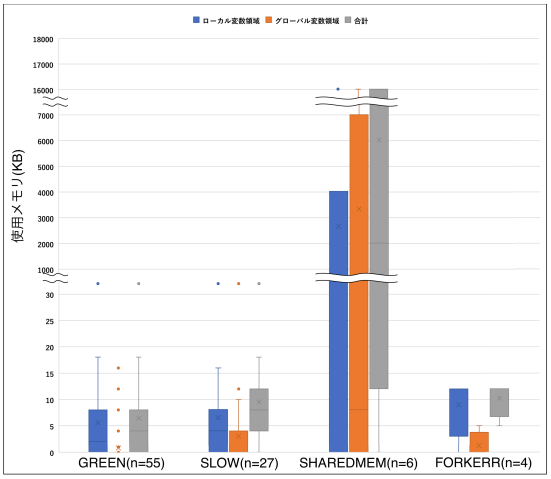

図 3　3.4 でシンボルとした変数のメモリサイズ

入力として与えられる変数の配列の長さを 10,000 未満と変更した．プログラムが静的に変数領域を確保していた場合については，変数領域を入力数と対応させた．

3.4.2　結果

プログラムに 3.2.1 項に記述した手順で KLEE を適用したところ 100 のプログラムのうち 55 が **GREEN**，27 が **SLOW**，4 が **FORKERR**，6 が **SHAREDMEM** となった．残りの 6 プログラムについては実行ごとに結果が大きく変化するため，定義しなかった．結果ごとのメトリクスの平均値を表 8 に示す．また，先ほどと同様にシンボルとした変数に与えたメモリの大きさの分布を図 3 に示す．さらに，コードカバレッジについて表 9 に示す．これらの図表については結果が変化する 6 のプログラムを除いた 94 のプログラムを対象としている．

また，本実験ではどのメトリクスがコードカバレッジに対して大きな影響を与えるのかを調べるため，命令カバレッジ，ブランチカバレッジを対象として線形回帰分析を行った．線形回帰分析の結果を表 10 に示す．

表 10 を見ると，コードカバレッジにはブロックの最大ネスト数のみが大きな影響を与えているということが見て取れる．以上の実験結果を踏まえて，結果と **RQ** との対応を下記に示す．

表 7　3. 3 で取得したメトリクスの平均値

	全体 (n=100)	GREEN (n=18)	SLOW (n=9)	SIGSEGV (n=30)	SHAREDMEM (n=38)
命令数	121.3	86.1	98.8	105.5	166.5
循環的複雑度の最大値	12.2	9.8	15.8	12.7	12.8
ブロックの最大ネスト数	3.8	3.9	4.0	3.7	3.8
実行メモリ (KB)	19,927.7	14,115.6	4,721.8	11,644.1	35,271.6

表 8　3. 4 で取得したメトリクスの平均値

	全体 (n=100)	GREEN (n=55)	SLOW (n=27)	SHAREDMEM (n=6)	FORKERR (n=4)
命令数	143.1	146.7	185.3	53.2	98.8
循環的複雑度の最大値	11.9	13.5	12.2	7.0	7.5
ブロックの最大ネスト数	3.8	3.8	3.8	3.0	3.8
実行メモリ (KB)	12,573.7	4,385.4	20,056.6	61,998.1	8,926.0

表 9　3. 3 でのコードカバレッジ

	50%を超えたコード数	50%を下回ったコード数
ICov	44	50
BCov	9	85

表 10　コードカバレッジの線形回帰分析結果

n=94	ICov	BCov
命令数	0.02	0.01
循環的複雑度の最大値	-0.41	-0.31
ブロックの最大ネスト数	5.54	5.78
シンボルのローカルメモリ	0.00	0.00
シンボルのグローバルメモリ	0.00	0.00
シンボルの合計メモリ	0.00	0.00
実行メモリ	0.00	0.00

RQ1

シンボルに与えるメモリサイズが大きい場合に KLEE が異常終了する．

RQ2

シンボルにグローバル変数を多用すると記号実行が低速になる．

RQ3

ブロックのネストが深いプログラムではテストケースのコードカバレッジが上昇する．

3. 5　考察

3. 4 節に記した各 **RQ** の回答に対する考察を行う．

まず，**RQ1** について考える．記号実行では実行経路に対応した経路制約を SAT Solver に与えることで変数の具体値を算出する．この時シンボルとした変数のメモリサイズが大きくなるとは SAT Solver に与える変数の数の増加を指し示している．SAT Solver の最悪計算量は変数の数に対して指数的に上昇する [4]．平均計算時間は指数ほど大きなものにはならないが，それでも変数の数の増加に伴い大きな計算量になっているということが予測される．結果としてプログラムの再帰回数が大幅に増えてしまい，SAT Solver は多量のメモリを利用し，エラーが発生したと考えられる．

次に，**RQ2** について考える．KLEE による記号実行では複数のスレッドでグローバル変数が共有される．したがって，変数のアクセスは場合により排他的になってしまう．結果としてローカル変数のみを対象としたプログラムよりも，動作が低速なものになっていると考えられる．

続いて，**RQ3** について考える．KLEE は，処理の高速化を図るため，可能な限り並列に記号実行を進めるように設計されている．そのため，記号実行を並列に実行可能であるプログラムでは高速になる．ブロックのネストが深いとはプログラム内に分岐が連続して存在することを指す．その結果，ブロックのネストが浅いプログラムより分岐が増えることが多く，実行経

路の数が増加する．実行経路が増加した結果として，記号実行の並列性が高まり，網羅的なテストケースが高速に生成されたと考えられる．

以上から，開発者や研究者が KLEE でカバレッジの高いテストケースを生成したい場合には，シンボルとなる変数のメモリサイズを小さなものとし，グローバル変数の多用は避けるべきということが言える．一方で，シンボルとした変数の多くがローカル変数で，ブロックのネスト数の多いプログラムでは高いコードカバレッジを誇るテストケース生成を期待できる．

また，表 10 を確認すると循環的複雑度はコードカバレッジにあまり影響しないということが見て取れる．一般的にプログラムが複雑になると人手での網羅的なテストケース，テストプログラムの作成は困難になる．しかし，KLEE であれば LLVM ビットコードを機械的に解釈してテストケースの作成が可能なため，複雑度の上昇に強い．したがって，KLEE は複雑すぎて人手によりテストケースの作成が困難な場合に大きな効果を発揮できると考えられる．

一方で，本実験ではシンボルのメモリサイズやプログラムの実行メモリの大きさなどのメモリに関する項目は，コードカバレッジに対して大きな影響を与えなかった．KLEE での変数の具体値の算出方法を考えると，これは非常に不可解な結果である．本実験で対象としたプログラムは競技プログラミングコンテストの提出プログラムを利用しており，その書き方や問題が与える制約は様々である．ある問題では数バイトのメモリ領域で入力を受け取ることが可能だが，あるプログラムでは数十キロバイトのメモリ領域が必要となってまう．結果としてシンボルのメモリサイズには大きな振れ幅が存在する．このため，シンボルのメモリサイズは線形的にはコードカバレッジとの関係を表現しきれなかった可能性があると考えられる．したがって，将来的にはメモリサイズとコードカバレッジの関係を様々な計算式に当てはめて近似し，どのような関係になるか調査する必要があると考える．

4 おわりに

本研究では競技プログラミングコンテストの提出プログラムを利用して，記号実行ツール KLEE を定量的評価を行った．研究を行うにあたり設定した **RQ** は以下である．

RQ1 KLEE の実行時エラーが発生しないかどうか．

RQ2 10 分間で 10 個以上の実行経路に対するテストケースが生成されるかどうか．

RQ3 生成されたテストケースのコードカバレッジが 50%を超えるかどうか．

AtCoder から抽出した 100 のプログラムに 10 分間の記号実行を行い，**RQ** に基づいて評価をしたところ，以下のことが明らかになった．

- シンボルとなる変数に与えるメモリサイズが大きい場合に異常終了する．
- シンボルにグローバル変数を多用すると実行が低速になる．
- プログラムのネストが深いとテストケースのコードカバレッジが上昇する．

この結果を受けて，KLEE を利用する開発者，研究者は KLEE を利用する際に，シンボルとなる変数のメモリサイズを小さなものとし，グローバル変数の多用を避けるべきということが示された．一方で，シンボルとする変数にグローバル変数が少なく，分岐や繰り返し処理の多いプログラムでは高いコードカバレッジを誇るテストケースの生成が可能であるということが明らかになった．

今後の課題として，他のプログラミングコンテスト運営サイトからプログラムを抽出し，同様の実験を行うということが挙げられる．各プログラミングコンテスト運営サイトでは作問者がそれぞれ異なるため，作成される問題には偏りがある．一方，本研究の目的は多用な特徴を持つプログラムに KLEE を適用し，KLEE の性能を評価することである．より偏りの少ないプログラムを対象として KLEE を評価するためには，各プログラミングコンテスト運営サイトからそれぞれプログラムを抽出し，KLEE の適用対象とする必要がある．

また，評価に利用する分析手法についても改めて検討する必要がある．本実験では線形回帰分析を利用して評価をしたが，シンボルとなる変数のメモリサイズやプログラムの実行メモリはコードカバレッジ

にほとんど影響を与えていなかった．しかし，SAT Solver の仕様を考えるとこれは不可解な結果である．このような結果となった原因にコードカバレッジとシンボルとなる変数のメモリサイズやプログラムの実行メモリとの間に線形回帰分析で表現できない関係があるということが考えられる．そこで，様々な計算式に当てはめて近似することで，シンボルとなる変数のメモリサイズやプログラムの実行メモリを始めとする各メトリクスと，コードカバレッジの関係をより正確に調査する必要がある．

参 考 文 献

[1] Cadar, C., Dumbar, D., and Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, *Proc. of OSDI*, Vol. 8, 2008, pp. 209–224.

[2] 片岡晃: ソフトウェア開発データ白書 *2018-2019*, 独立行政法人情報処理推進機構, 2018, pp. 178–188.

[3] 宋剛秀, 番原睦則, 田村直之, 鍋島英知: SAT ソルバーの最新動向と利用技術, コンピュータソフトウェア, Vol. 35, 日本ソフトウェア科学会, 2018, pp. 72–92.

[4] 井上克己, 田村直之: SAT ソルバーの基礎, 人工知能学会誌, Vol. 25, 人工知能学会, 2010, pp. 57–67.

[5] 上原忠弘: シンボリック実行を利用した網羅的テストケース生成, *FUJITSU*, Vol. 66(2015), pp. 34–40.

[6] 徳本普, 上原忠宏, 宗像一樹, 菊地英幸, 江口享, 石田晴行, 馬場匡史: C/C++シンボリック実行ツール KLEE の適用と CPPUNIT 連携ツールの開発, *ESS*, (2011).

[7] 渡部有隆: オンラインジャッジの開発と運用-Aizu Online Judge-, 情報処理, Vol. 56, 情報処理学会, 2015, pp. 998–1005.

[8] 中野隆司, 田中裕大, ダン ティ ホンイエン: ソフトウェアのテスト工数・期間を削減するためのシステムテスト自動化技術, 東芝レビュー, Vol. 73, 東芝, 2018, pp. 40–44.