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

徳本 晋[†] 上原忠弘[†] 宗像一樹[†]
菊地英幸[†] 江口亨[††]
石田晴幸[††] 馬場匡史[††]

KLEE は LLVM 中間コードを対象としてシンボリック実行をすることができるツールである．C/C++で書かれたプログラムは llvm-gcc や clang などのコンパイラで LLVM ビットコードにコンパイルできるため，C/C++プログラムの検証ツールとして KLEE は期待されている．今回，2 つのプログラムについて KLEE を適用し，それにより検出したバグとその特徴を紹介するとともに，KLEE のバグ検出能力について考察する．また，他の商用 C/C++検証ツールで対象プログラムを解析したときの結果と，KLEE を使った場合との比較について報告する．さらに KLEE 適用時の課題を解決すべく，KLEE で生成したテストケースを CPPUNIT のテストコードへ変換するツールについても報告する．

The Application of C/C++ Symbolic Execution Tool KLEE and the Development of KLEE-CPPUNIT Cooperation Tool

Susumu Tokumoto,[†] Tadahiro Uehara,[†] Kazuki Munakata,[†]
Hideyuki Kikuchi,[†] Toru Eguchi,[††] Haruyuki Ishida[††]
and Masafumi Baba[††]

KLEE is a symbolic execution tool for LLVM intermediate code. It is regarded as a C/C++ program verification tool, because the program written in C/C++ can be compiled to the LLVM bit code by compilers such as llvm-gcc and clang. We applied KLEE to two programs, and show the detected bugs and its characteristics, and we study the bug detection ability of KLEE. Moreover, we reports on the comparison of the result that the programs are analyzed with other industrial C/C++ verification tools and KLEE. In addition, to solve the problem when KLEE is applied, we developed the tool that converts the test case generated with KLEE into the test code of CPPUNIT.

1. はじめに

プログラムのバグを大別すると，実行できない操作を試行した時に起こる実行時エラーと，実行した結果が仕様と一致しない論理エラーに分けられる．実行時エラーを検出するツールとしては静的コード解析ツールが存在するが，false positive が多いなどの問題がある．また，論理エラーを検出するツールとしては単体テストツールが存在するが，網羅性の高いテストコードを記述するには多くのコストがかかる．

シンボリック実行ツール[1)2)]は，実行時の分岐条件を元に変数のとり得る範囲の場合分けを行いながら，網羅的にパスを実行することで，false positive のない実行エラーの検出と，パス網羅のテストケースの自動生成ができる強力な検証用ツールである．KLEE[3)4)]は LLVM(Low Level Virtual Machine) ビットコードを対象としてシンボリック実行を行うツールである．C/C++で記述されたプログラムは llvm-gcc や clang などのコンパイラで LLVM ビットコードにコンパイルできるため，KLEE は C/C++プログラムのシンボリック実行ツールとして期待が高い．

今回，KLEE の実行エラー検出能力を調査すべく，ネットワーク性能測定用のオープンソースのプログラムである Iperf と，自律走行ロボットの制御プログラムの 2 つのプログラムに対して KLEE 適用を試行した過程とその結果について報告する．また，これらのプログラムに対し，2 つの商用静的コード解析ツールにて解析を行い，KLEE の解析結果との比較と考察を

† 株式会社富士通研究所
Fujitsu Laboratories Limited
†† 株式会社富士通コンピュータテクノロジーズ
Fujitsu Computer Technologies Limited

行った．さらに，効率的に論理エラーの検出をするために，KLEE で生成したテストケースで単体テストを行う CPPUNIT のテストコードを生成するツールを開発し，通常の CPPUNIT を用いた単体テストとの比較をすることでツールの効果を評価した．

2. KLEE 概要

KLEE は LLVM ビットコードを読み取り，シンボリック実行を行うツールである．ここではまずシンボリック実行と LLVM について説明する．

2.1 シンボリック実行

シンボリック実行は，入力をシンボルという具体値を伴わない値として扱う実行方式である．ここではプログラム中でシンボルを表す変数をシンボル変数と呼ぶ．シンボル変数を含む条件分岐に対してそれぞれのパスを実行し，最終的にパスを網羅するまで実行を繰り返す．また，1 つのパスにおけるシンボルの取り得る条件をパス条件と呼び，SAT/SMT ソルバを用いてパス条件を満たす具体値を得ることで，そのパスを実行するテストケースを生成できる．つまりすべてのパス条件を解くことで，パス網羅のテストケースを生成できる．もしパス条件を満たす解が存在しない場合，そのパスは実際には辿りつかないパスになるので，実行をせず，テストケースも出力されない．

具体的にリスト 1 を例として説明する．

リスト 1　bad_abs

```
1  int bad_abs(int x){
2      if(x == 123)
3          x++;
4      if(x < 0)
5          x = -x;
6      return x;
7  }
```

引数 x をシンボル Sym_x とした場合，2 行目と 4 行目にシンボル変数を含む分岐があるので，パス条件は以下になる．

(1)　$(Sym_x = 123) \wedge (Sym_x < 0)$
(2)　$(Sym_x = 123) \wedge \neg(Sym_x < 0)$
(3)　$\neg(Sym_x = 123) \wedge (Sym_x < 0)$
(4)　$\neg(Sym_x = 123) \wedge \neg(Sym_x < 0)$

しかし，(1) についてはこれを満たす Sym_x が存在しないため，実行は行われず，テストケースも出力しない．よって図 1 のように 3 つのパスが実行され，それぞれに対応するテストケースが生成される．

2.2 LLVM

LLVM は JavaVM のようにビットコード（中間コード）をマシン向けコードに変換して実行する仮想機械である．LLVM ビットコードはプログラミング言語から

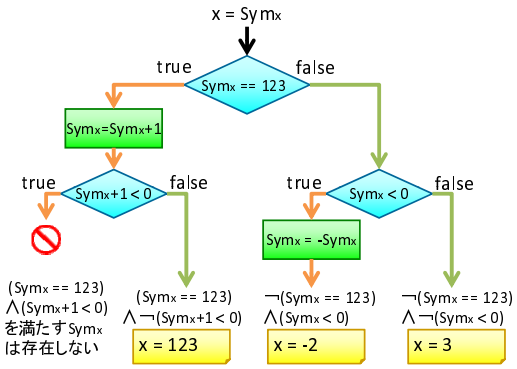

図 1　パスと対応するテストケース

ら独立した命令セットと型システムを持っているが，LLVM ビットコードを生成可能なコンパイラの中で現在最も成熟しているのが C/C++コンパイラである LLVM-GCC と CLANG である．KLEE では仮想機械と同じように LLVM ビットコードの命令を解釈し，実行を進める．ただし，シンボル変数を含む分岐命令であれば，パス条件が充足可能なことを確認し，その時点の状態を保存した上で，片方のパスの実行を進め，そのパスが終端まで辿りついた後にもう一方のパスの実行を進める．

3. 対象プログラムについて

ここでは KLEE を適用したプログラムのうち，実行バグを検出できた 2 つのプログラムについて紹介する．

3.1 Iperf

Iperf[5] はオープンソースの TCP/UDP の帯域測定ツールである．測定したい通信路の両端の端末のコマンドライン上から起動し，クライアントからサーバへ可能な限り高速にデータを送信する．クライアント，サーバの指定はオプションで行い，また TCP/UDP の設定やチューニングなどもオプションから指定できる．

iperf-2.0.5 は C/C++で記述されており，全体では約 15KLOC である．機能を大別するとサーバ機能とクライアント機能に分けられるが，今回，その中のサーバ機能に関する処理について検証を行った．

3.2 自律走行ロボット

ET ソフトウェアデザインロボットコンテスト（以下，ET ロボコン）[6] では，共通の設計の LEGO Mindstorms NXT ロボットに，各チームそれぞれで設計したプログラムを載せ，その性能と設計したモデルの品質を競う．

2009 年と 2010 年の大会に参加した自社の BERMUDA

というチームでは，モデル駆動型開発を行っており[7]，その自動生成コードとハンドコーディングのコードを対象として検証を行った．規模としては約7KLOCで，そのうち約75%が自動生成のコードである．コード生成ツールは自社開発した独自ツールであるが，コード生成ツールの入力となるモデルの記述は商用モデリングツールを用いている．

4. KLEE適用

4.1 適用の準備

前章で述べた各プログラムに対して，KLEEを適用を行った．適用の手順としては以下になる．

(1) 対象となるプログラムを特定する
(2) シンボルとする変数を特定する
(3) （ない場合は）スタブ，ドライバを用意する
(4) コンパイルを行う
(5) KLEEで解析処理を行う

対象となるプログラムを特定する際に，解析の準備に非常に多くの工数がかかるものや，解析をしたところで得られる効果が薄いものもあるため，注意しなくてはならない．以下のようなものがそのようなプログラムと考えられる．

- バイナリ提供のみのライブラリやフレームワークを多用するもの
- 入力の値によって条件分岐しないもの
- 規模が大きく，適当な大きさに分割不可能なもの
- アセンブリ言語を多用しているもの

KLEEはLLVMのビットコードに対して解析を行うため，バイナリ提供のみのライブラリやフレームワークのシンボリック実行は不可能である．よって該当部分のドライバやスタブを用意しなければならない．標準Cライブラリや POSIX システムコールについてはKLEE用に改造したライブラリが用意されている．また標準C++ライブラリに関しても開発が行われている[8]．一方，並列実行のためのシステムコール，ライブラリ（fork,pthreadなど）は一部の研究者が開発を進めているもの[9]，現状ではサポートしておらず，今後の開発が期待される．

同じ理由で，アセンブリ言語に関しては高級言語で記述されたスタブへの置換などでLLVMビットコードへの変換ができないと，KLEEの実行ができない．

シンボル変数の選択においても，適切な変数を選ばないと分析時間が非常にかかる場合や，テストケースのバリエーションができない場合がある．具体的には以下のような変数の場合になる．

- ループ中の条件分岐に含まれる変数
  例えば，以下のようなint型変数内で1が立っているビットを数えるようなプログラムでは2^{32}個(int型が32ビットの場合)の実行パスつまりテストケースが生成しようとするが，現実的な時間では実行が終了しなくなる．

リスト2 countbit

```
1 int countbit(int x){
2     int cnt = 0, int i = 0;
3     while(i++ < sizeof(int)*8){
4         if((x >> i) & 1) cnt++;
5     }
6     return cnt;
7 }
```

- ポインタ
  ポインタをシンボルとした場合，すべてのアドレス空間を探索することとなるため，不正なアドレスへのアクセスが発生する可能性がある．
- 条件分岐に影響しない変数
  シンボル変数を含む分岐条件を元にテストケースを生成するため，条件分岐に影響しない変数をシンボル変数に選択してもテストケースのバリエーションができない
- 浮動小数点変数
  現状ではKLEEでサポートしていないため，具体値に置き換えられて実行される．浮動小数点をサポートしたKLEEの研究は行われている．[10]

4.2 事例におけるスタブ・ドライバの作成

Iperf，自律走行ロボットともに図2のように動作設定やコマンド引数のような静的情報をテスト対象の呼び出し元から与え，センサ情報やパケットのような動的情報についてはテスト対象から呼ばれるAPIやシステムコールから与えられている．よって，静的情報はドライバでシンボルとして用意し，動的情報はスタブでシンボルとして用意する．動的情報取得の関数は複数回呼ばれる可能性があるため，状態爆発を起こさないよう繰り返し回数を絞り，呼び出される度にシンボルを作るようにして，テスト対象の状態をより網羅するようにした．

4.3 KLEEによる解析

KLEEの解析は次の実行エラーを検出できる．

- メモリ不正アクセス
- ゼロ除算
- 非ヒープ領域のメモリ解放
- 権限を持たない読み書き
- assert,abortによる終了

今回，Iperfと自律走行ロボットの2つのプロジェクトにおいて，メモリ不正アクセスのバグを検出した．

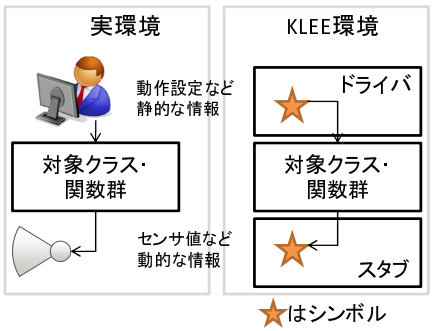

図 2 スタブ・ドライバの配置

4.3.1 Iperf で検出したバグ内容

Iperf の処理は主にサーバ処理とクライアント処理に分けられるが，今回はサーバ処理に対してシンボリック実行を行った．ドライバではコマンド引数など動作設定に関する変数をシンボルとし，システムコールをスタブ化して返却値などをシンボルとした．

結果として，1 件のメモリ不正アクセスのバグを検出した．図 3 のように，UDP 通信のオプション-u と実行結果レポートを表示しないオプション-xCD を同時に指定した場合に，関数 InitReport ではレポート用データ領域のメモリを確保しないにもかかわらず，メソッド write_UDP_AckFIN においてレポート用データ領域のメンバにアクセスする処理が含まれていた．

通常 Iperf を利用する場合，測定結果が重要であり，レポートを出さないというコマンド引数が使われることが少ないため，テストケースの考慮漏れが起きていたことが考えられる．また，関数・ファイルをまたがってメモリ確保とメモリアクセスを行っているため，レビュー時に目視で追っていくことが難しかったことも原因の一つと考えられる．このようなレアケースの考慮漏れや目視で追うことが難しいプログラムに対して KLEE は効果を発揮することがわかった．

4.3.2 自律走行ロボットで検出したバグ

ET ロボコン 2010 では前年の大会で開発した資産を流用し，派生開発を行った．追加や変更に影響するモジュールに関してはトレーサビリティマトリクスなどで確認を行った．しかし，プログラム実行中に異常終了してしまうという現象が起こっていたため，KLEE による解析を行い，その現象の原因を探ることにした．

センサ値取得 API を，センサ値をシンボルとして返すスタブに置き換え，ドライバでは状態遷移を網羅するために必要な回数の繰り返し処理を行うようにした．

その結果，流用元のモデルから生成されるソースコードを端としたメモリ不正アクセスが起こっていることが判明した．以前は図 4 左のように列挙型 JudgeType の先頭は TIME_JUDGETYPE=0 であったが，モデリングツールのバージョンアップにより，図 4 右のように列挙型 JudgeType がアルファベット順に並び替えられ，新しく先頭になった ABSOLUTE_JUDGETYPE と後方に回った TIME_JUDGETYPE=0 の 2 つが 0 となっていた．そのことにより，リスト 3 のように，入力変数 judgeType が TIME_JUDGETYPE のときのみ後続の処理を続けるプログラムにおいて ABSOLUTE_JUDGETYPE の場合も後続の処理を続けてしまい，その結果メモリを確保していない領域にアクセスしてしまっていた．

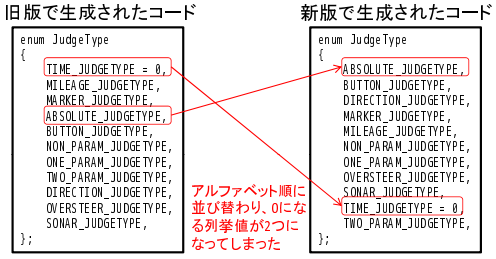

図 4 モデリングツールで生成されたコード

リスト 3 TimeJudge::Run()

```
1  void TimeJudge::Run()
2  {
3      if( (Rte_Receive_in(judgeIF) != RTE_E_OK) ||
4          (judgeIF.judgeType != TIME_JUDGETYPE) )
5          return;
6
7      // TIME_JUDGETYPEのときはParam1へのキャスト可能
8      unsigned int timeout =
9          (unsigned int)((Param1*)judgeIF.param)->param;
```

モデル駆動型開発において自動生成されたコードは可読性を考慮されていない部分も多く，人手による要因分析からテストケースを作成することが困難だが，シンボリック実行によって自動で網羅的に実行エラーを探索し，高カバレッジのテストケースを生成することで，そのような問題も解決できることがわかった．

5. 静的コード解析ツールとの比較

前章で紹介した KLEE を適用し検出したバグについて，2 つの商用静的コード解析ツールで検出可能か実験を行った．2 つとも世界的なシェアとしても上位に入るものである．

一方の商用解析ツール（以下ツール A とする）はプログラムのバグだけでなく，コーディングスタイルの指摘やリファクタリングの推奨やメトリクスも出力することを特徴としている．もう一方のツール（以下ツール B とする）はオーバーフローやメモリリーク

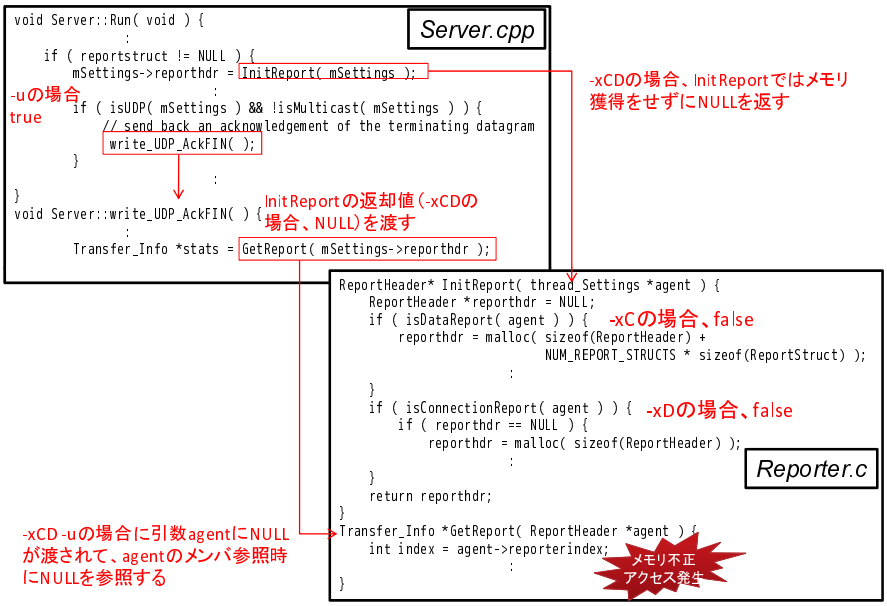

図 3 Iperf で検出したバグ

の検出だけでなく，並列プログラムにおける競合など多様なバグを検出できることを特徴としている．

5.1 Iperf の静的コード解析

Iperf のコード解析において，ツール A では 33 件の指摘があった．主な指摘内容の種類は以下になる．

(1) バッファオーバーフローの可能性がある
(2) フォーマット文字列が外部注入による脆弱性の可能性がある
(3) メモリ解放をしない可能性がある
(4) NULL を返す可能性のある関数で NULL の確認をしてない
(5) 到達不能コードの存在

(1) に関しては同一原因のバグが 4 件指摘されていた．これは浮動小数点変数による分岐を含むバグであったため，KLEE では検出できなかったが，ツール A では検出可能だった．(2) に関する指摘は 10 件あったが，どれも脆弱性になることはないものだった．(3) に関しては 3 件あったが，単一関数内での判断でありプログラム全体としては問題ない個所であった．(4) に関しては 2 件あり，それぞれ malloc と localtime の返却値の NULL チェックをしていないことの指摘だった．KLEE ではこれらの関数をスタブで用意することでメモリ確保失敗時の処理についても検出可能だが，今回の検証の範囲からは外していた．

次にツール B によるコード解析だが，15 件の指摘が出力された．指摘内容の種類は以下になる．

(1) sprintf(),strcpy() などを snprintf(),strncpy() などへの置き換えを推奨
(2) NULL 参照の可能性がある
(3) 排他制御ロックが解放されない可能性がある
(4) バッファオーバーフローの可能性がある

指摘項目の大半が (1) に関してで，すでに考慮済みの値を使っているため，問題が起こることはない．(2) に関しては 2 件あり，1 件は実際には問題が起きない指摘で，もう 1 件は KLEE で検出したバグと同じポインタに対しての指摘だったが，NULL 参照が起きうると指摘されていた個所は異なり，問題がない個所であった．(3) に関しては，KLEE ではサポートされていない並列性に関する指摘であるが，2 件の指摘のうち，1 件はバグであり，もう 1 件は問題のない指摘だった．(4) の指摘はツール A の (1) の指摘と同内容であり，ツール B でも検出可能だった．

指摘数とバグ数と KLEE で検出したバグの検出可否について表 1 にまとめる．

表 1 Iperf における静的解析ツールとの比較

	指摘数	バグ数	KLEE 検出バグ
ツール A	33 件	3 件	検出不可
ツール B	15 件	2 件	検出不可

5.2 自律走行ロボットの静的コード解析

自律走行ロボットのコード解析において，ツール A では 381 件の指摘が出力された．指摘内容の種類は以下になる．

(1) printf フォーマットと対応する変数の型の不一致

(2) 到達不能コードの存在
(3) コンストラクタ内で初期化されていないメンバ変数が存在する
(4) int から short へのキャストにおける精度の消失

このうち (1),(2) はスタブコードにおける指摘で，(3),(4) に関しても考慮済みで問題は起こらない指摘であった．またこの中に KLEE で検出したバグは指摘されなかった．

次にツール B によるコード解析だが，16 件の指摘が出力された．指摘内容は以下になる．

(1) 通らないパスが存在する
(2) コンストラクタ内で初期化されていないメンバ変数が存在する

(1) に関しては自動生成された値を条件としたため分岐しないパスがあるのが理由だが，モデルの変更による影響を減らすための対応なので問題ない．(2) に関してはツール A の (3) と同じである．

指摘数とバグ数と KLEE で検出したバグの検出可否について表 2 にまとめる．

表 2 自律走行ロボットにおける静的解析ツールとの比較

	指摘数	バグ数	KLEE 検出バグ
ツール A	381 件	0 件	検出不可
ツール B	16 件	0 件	検出不可

5.3 解析能力の比較

今回シンボリック実行ツールである KLEE で検出したバグについて，2 つの静的コード解析ツールでは検出することができなかった．

その理由として，コンテキストの依存性の違いがある．静的コード解析ツールでは実際に実行させることなく，関数，メソッドごとに独立して解析を行い，どのようなコンテキストの上で実行されるかについては考慮しない．そのため解析時間はそれほど長くならず，メモリ消費も少ないが，コンテキストを絞り込めないので false positive が多くなる．一方，シンボリック実行ツールでは実際に実行することでエントリポイントから状態の変化を追い，関数・メソッドの解析をそこまでのコンテキストの上で行う．そのため解析時間は比較的長く，メモリを多く消費するが，あり得ないコンテキストで解析することはないため false positive はない．

今回 KLEE で検出できたバグは，関数・メソッド間にまたがる処理におけるメモリ不正アクセスであり，1 つ 1 つの関数・メソッドを解析するだけでは検出が不可能である．このように関数・メソッド間にまたがる処理で起きるバグはシンボリック実行を始めとする動的コード解析であれば検出できる可能性が高い．シンボリック実行であれば，適切なシンボルの設定により可能性のあるコンテキストを漏れなく用意し，網羅的な解析をすることができる．

逆に KLEE で false negative になるケースとしては，浮動小数点変数の分岐や並列処理といった KLEE でサポートしていない機能に関わるバグや，適切なシンボルを用意できず網羅できないパス上のバグがある．これらは KLEE における今後の課題であると考えられる．

5.4 解析準備工数の比較

Iperf の事例において，解析準備に必要な工数を計測したところ，KLEE では 1 人日，ツール A とツール B では 0.2 人日となった．

KLEE ではシンボル変数の特定のための要因分析や，ドライバ・スタブの作成や，LLVM コンパイル用 Makefile の作成など，解析準備により多くの時間が必要となる．また状態爆発が起こらないようにすることや，テストケースにより多くのバリエーションができるようにすることなどの試行錯誤も必要である．

一方，静的解析では，必要なのは解析設定ファイルを作成することくらいで，要因分析やドライバ・スタブの用意はほとんどの場合で不要なため，準備にはそれほど多くの時間がかからない．

6. CPPUNIT 連携ツール KUCHEN の開発

前章まで KLEE の実行エラー検出能力について示してきたが，論理エラーについては KLEE のみで自動的に検出することができない．なぜなら，論理エラーはテストケースとその期待値を考えた上でテストコードを書き，テストツールで実行結果と期待値を照合することで検出することができるが，KLEE では網羅的なテストケースの生成を行えても，その期待値との照合を支援する機能はないためである．また KLEE で生成した高カバレッジのテストケースはその数が膨大になるため，実行するテストコードを人力で記述するには多くの工数を必要とする．今回，我々は KLEE のテストケースが埋め込まれた CPPUNIT 用テストコードを生成するツール KUCHEN を開発し，高カバレッジのテストを少ない工数で行うことで効率的な論理エラーの検出を可能にした．

6.1 機 能 概 要

図 5 のようにテスト対象の C++のクラスに対して，自動で生成したテストケースを CPPUNIT のテストコードに埋め込む形でコード生成を行う．実行結果と

比較する期待値についてはユーザが生成コードに入力を行う．

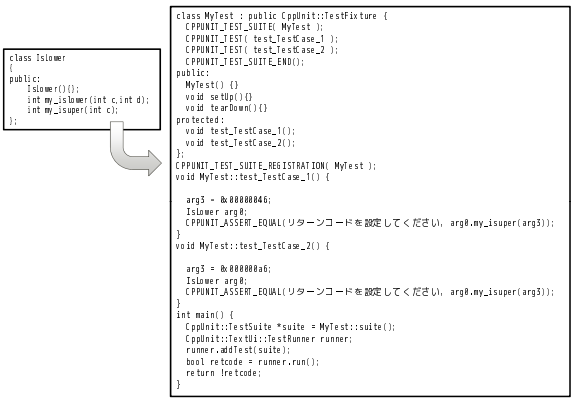

図 5　KUCHEN による CPPUNIT コードへの変換

6.2　設 計 概 要

テスト対象クラスから CPPUNIT のソースコード生成までの流れを図 6 に示す．

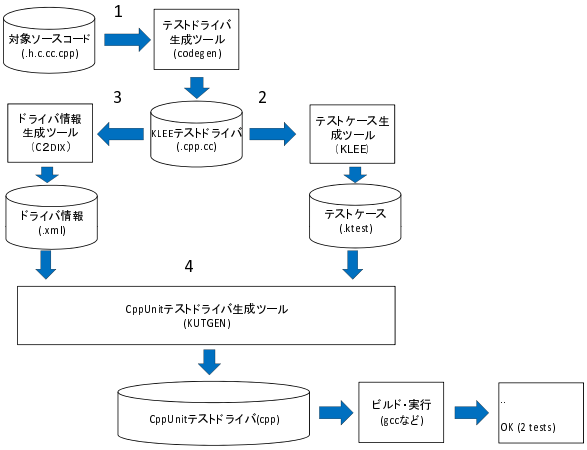

図 6　対象ソースコードから CPPUNIT コードへの流れ

まずテスト対象クラスを含むソースコードを構文解析し，クラス中のメソッドの引数をシンボルとしてそのメソッドを実行する KLEE テストドライバを生成する（図中 1）．次に KLEE テストドライバと対象ソースコードを LLVM 中間コードへコンパイルして，KLEE でシンボリック実行を行い，テストケースを出力する（図中 2）．そして，KLEE テストドライバを構文解析し，そこからテスト対象としたメソッドやシンボルなどのドライバ情報を XML として格納する（図中 3）．最後にそのドライバ情報の XML とテストケースを元に CPPUNIT テストコードを生成する（図中 4）

現状の KLEE ではシンボル変数を含む分岐に対してパス網羅のテストケースを生成するため，テストケース数が膨大になり，すべての期待値を入力するのは非常に困難である．そこでテストケースをパス網羅から条件網羅や複合条件網羅へ絞り込むツール[11] を用いて，テストの効果をできるだけ損なわないようにテストケース数を人間が扱える現実的な数まで削減する．

6.3　評　　価

今回，実製品で使われている文字列処理を行う 0.5KLOC のソースコードに対して，KUCHEN により CPPUNIT コードを生成し，人力で CPPUNIT コードを作成する場合との比較を行った．表がその結果になる．

表 3　KUCHEN と人力による CPPUNIT コード作成の比較

	KUCHEN	人力
テストケース	38 件 (868 件から絞り込み)	20 件
命令網羅率	92.2%	82.1%
工数	1 人日	5 人日

今回，KLEE で生成されたテストケースは 868 件になったが，条件網羅への絞り込みを行うことで 38 件まで減らし，CPPUNIT コードを生成した．KLEE ドライバの修正や KLEE 解析時間や期待値の埋め込みなどの工数があるものの，KUCHEN では人力よりも少ない工数でテストケースを作成することができた．また条件網羅への絞り込みは，命令網羅率を損なうことなく行うことができ，人力よりも KLEE では高い命令網羅率を達成できた．

7.　関 連 研 究

C のシンボリック実行ツールとしては，KLEE の他にも CREST[12] や CUTE[13] がある．これらは KLEE とは異なり，異なるパスを選ぶような具体値での実行を繰り返すコンコリック実行という手法をとっている．今回，我々が KLEE を採用した理由としては，オープンソースで公開されていること，実行オプションの豊富さや周辺ツールの充実などツールの成熟度が他のツールに比べ高いこと，LLVM 上で実行するため環境に依存せずに使えることが挙げられる．

Emanuelsson らは商用の静的コード解析ツールの比較を行っている[14]．静的コード解析の比較する軸として実行エラー検出能力と精度と解析時間を挙げ，主に 3 つのツールについての特徴を挙げた上で，それらの比較と評価を行っている．我々はシンボリック実行と静的コード解析との比較のため，比較の観点が若干

異なるが，我々が用いた 2 つの静的コード解析ツールの特徴を知る上で有意な情報となった．

Cadar らは近年開発がすすめられているシンボリック実行ツールをまとめ，それぞれの特徴を述べている[2]．しかし，内容としては各ツールの紹介に止まり，ツールの解析能力などの比較はされていない．

Yu らは MSC-51 向けの中間言語を生成し，シンボリック実行を行う組込みシステム向けの手法を示した[15]．我々が採用した KLEE では LLVM 上で実行するため，特定の環境に依存せずに利用可能である．ただし，アセンブリ言語を含むなど実行環境に依存したソースコードを記述する場合は Yu らの手法は有効と考える．

8. ま と め

本論文では Iperf と自律走行ロボットのプログラムを題材として KLEE の適用方法と検出したバグとその特徴を示した．また，これらのプログラムに対して 2 種類の商用静的コード解析ツールで解析し，KLEE で検出したバグについてこれらのツールでは検出不可能であったことを紹介した．さらに KLEE と静的コード解析ツールのそれぞれの解析能力と解析準備工数について分析と考察を行った．そして，KLEE で生成したテストケースを設計における期待値と比較するべく，テスト対象のソースコードから CPPUNIT のテストコードを生成するツールを開発し，その評価を行った．

今後は KUCHEN の課題の解決と，より大きいプロジェクトへの適用を進めていくことを考えている．KUCHEN の課題としては，現状では期待値と比較する変数をメソッド返却値としているが，本来はコンテキストに則したものを選ぶようにできるとよいと考えている．また，現状では人手によって入力している期待値を，リグレッションテストのように大半のものは実行結果と期待値が等しいものについては自動で入力できることが望ましいと考えている．

参 考 文 献

1) King, J. C.: Symbolic execution and program testing, *Commun. ACM*, Vol. 19, pp. 385–394 (1976).
2) Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N. and Visser, W.: Symbolic execution for software testing in practice: preliminary assessment, *Proceeding of the 33rd international conference on Software engineering*, pp. 1066–1071 (2011).
3) Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted Automatic Generation of High-Coverage Tests for Complex Systems Programs, *USENIX Symposium on Operating Systems Design and Implementation* (2008).
4) *The KLEE Symbolic Virtual Machine*. http://klee.llvm.org/.
5) *Iperf*. http://sourceforge.net/projects/iperf/.
6) ET ロボコン公式サイト. http://www.etrobo.jp/.
7) 徳本晋, 江口亨, 辻村浩史, 村上亮, 伊澤松太朗, 松本博郎, 山村健太郎: 組込みシステムにおける複数の DSL を用いたプロダクトライン開発, 情報処理学会第 72 回全国大会 (2010).
8) Li, G., Ghosh, I. and Rajan, S. P.: KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs, *Proceeding of the 23rd International Conference on Computer Aided Verification (CAV) 2011* (2011).
9) Zamfir, C. and Candea, G.: Execution synthesis: a technique for automated software debugging, *Proceedings of the 5th European conference on Computer systems*, pp.321–334 (2010).
10) Collingbourne, P., Cadar, C. and Kelly, P. H.: Symbolic crosschecking of floating-point and SIMD code, *Proceedings of the sixth conference on Computer systems*, pp. 315–328 (2011).
11) Munakata, K., Fujiwara, S., Tokumoto, S., Maeda, Y. and Uehara, T.: Coverage-Oriented Test Case Selection based on Path Conditions of Symbolic Execution (2011). to be published.
12) Burnim, J. and Sen, K.: Heuristics for Scalable Dynamic Test Generation, *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 443–446 (2008).
13) Sen, K., Marinov, D. and Agha, G.: CUTE: a concolic unit testing engine for C, *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 263–272 (2005).
14) Emanuelsson, P. and Nilsson, U.: A Comparative Study of Industrial Static Analysis Tools, *Electron. Notes Theor. Comput. Sci.*, Vol. 217, pp. 5–21 (2008).
15) Yu, H., Song, H., Xiaoming, L. and Xiushan, Y.: Using Symbolic Execution in Embedded Software Testing, *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, pp. 738–742 (2008).