安全系組込ソフトウェア開発におけるユニットテストの効率化 ～Concolic Testingの活用事例～

岸本 渉
株式会社デンソー
wataru_kishimoto@denso.co.jp

要旨

機能安全など，安全系のソフトウェアに対する詳細なテスト要件は *MC/DC* カバレッジなど具体的な要求として浸透し始めている．人手に頼ったテストでは抜け漏れ等の問題と，コスト増の問題が生じ対応できないため自動化が進んでいる．しかし，現有のツールでは *C0*，*C1*，*MC/DC* カバレッジ *100* %を達成することが出来ず，その部分を人手で対処するため大きな損失が生じている．

本稿では，ソフトウェア工学における最新の研究成果が反映しているオープン系のツール *Concolic Testing* を活用してこの問題を解決したので報告する．オープン系のツールは導入時に，技術的なスキルが求められるが，様々な工夫を行うことが出来るので，柔軟に対応し，導入時の問題を解決したので報告する．

1. はじめに

我々は，センサで車両の状態を検知し，その結果を用いて安全系装置を制御する組込ソフトウェアの開発を担当している．このシステムは，車種やグレードにおけるオプション搭載ではなく標準搭載が進み，他サプライヤとの価格競争が激化している．高い品質と自動車機能安全規格 ISO26262 に準拠するなど安全性を担保しつつ，一層のコスト削減が求められている．さらに，車両の安全系機能が多様化し，従来以上に開発短納期化の要求が強くなり，コスト削減以上に開発サイクルの短期化が求められている [1].

このような状況に対応するには，従来のテスト方式を基に要員の増強や方式改善では限界がある．当然，自動化を進める必要があるが，従来の自動化のように人手で出来ることをスクリプトや記述言語で開発，蓄積して実行を自動化する方式ではやはり限界がある．

ソフトウェア工学では，近年，モデル検査の技術を拡張し，ソースコードを探索する技術が急激に発達している [2]．静的解析では，有力なプロプライエタリ系のツールが出現し，この分野では人手による方式を完全に駆逐しつつある．しかし，仕様ベースのテストについては，様々な研究が行われているが，有力な自動化ツールは存在しない．

現状では，我々が求める自動化を達成できる統合ツールは存在しないが，部分的には利用できる技術が出現し，それらの技術は現在も発達中である．具体的には，Symbolic Execution や Concolic testing など，ソースコードの高度な解析を行う技術を指す [3, 4, 5]．この分野では，NASA の JPF/SPF や LLVM/KLEE や CIL/CREST などオープン系のツールが先行しており，これらを超えるプロプライエタリ系のツールは出現していない [6, 7, 8].

本論文では，高度な解析技術を実際のユニットテストに活用する試みについて報告する．この種の技術は，最先端のオープン系技術を使いこなす高度なエンジニアを育成しないと利用することが困難である．将来に向けて新たなソフトウェア生産技術を構築する試みとして評価を兼ねて行った．

本論文の構成と主な内容は以下の通りである．2 章では，現行のユニットテスト方式における課題を明確にし，3 章では，その対策と導入するツールについて説明する．4 章では，我々のソースコードに対してツールは有効に機能するのかを確認した．その方法は，あるプロジェクトのソースコードのなかからサンプリングしたものを評価用に書きなおしたコードを使って行った．その結果，

ツールを使うための工夫を行えば十分使えることが判った．5 章では，実際にプロジェクトにおけるユニット群に対し，現行のツールでは MC/DC カバレッジ 100%が未達であるユニットに適用し，達成できることを確認できた．6 章では，この結果からテスト方式を変更することについて検討した．

2. 現行ユニットテスト方式の分析

安全系組込ソフトウェアは，自動車機能安全規格 ISO26262 に準拠することが求められている．そのため，我々の職場では，ユニットテストにおいて，実装マイコンコードを評価対象としてマイコンシミュレータでユニットテスト実行を行う仕組みを持つツールと，そのツールを効率よく用いるための内製ツールを採用している（図 1）．このユニットテストツールはソース解析を行い C0，C1，MC/DC カバレッジを 100%達成するテストケースを生成する機能も持っている．

これらのツールを用いて大部分の作業を自動化しているが，実際のところ C0，C1，MC/DC カバレッジが 100%未満のテストケースが生成される場合がある．安全系のテスト対象ユニット，すなわち ASIL A 以上が割り当てられたユニットでは，人手により C0，C1，MC/DC カバレッジを 100%達成するテストケースを追加する作業が発生する．これは，単にカバレッジが悪い所を見れば追加できる容易な作業ではなく，カバレッジが達成できていない箇所に至る制御経路を解析する必要がある．そのため，安全系のユニットテストは，多くの工数を必要とし，生産性を低下させ，納期を圧迫している．

3. 対策と CREST の導入

3.1. 対策の考え方

前章で分析した通り，現在使用しているユニットテストツールだけでは十分なテストケースの生成ができない．安全系以外のユニットで C0，C1，MC/DC カバレッジが 100%未満のユニットに対しては，実機を用いた統合テストにおいてユニットテストレベルの詳細なテストを大きな工数をかけて行っており，システムテストを含めたテスト工程全体の効率化の妨げとなっている．

したがって，十分なテストケースが自動で生成できるようになれば，MC/DC カバレッジ 100%のユニットテスト対象ユニットを安全系かどうかに関わらず全ユニットに拡大することができる．それによって，統合テストにおけるユニットテストレベルの詳細なテストが不要となり，生産性と品質の両方を向上できる．

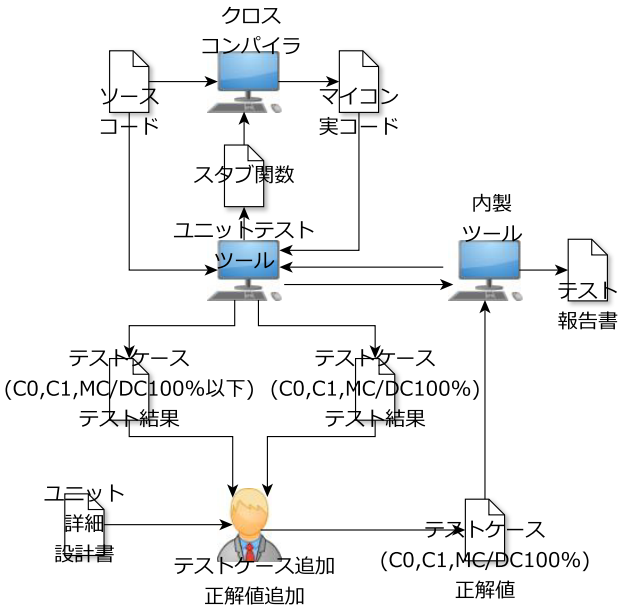

図 1. 現行ユニットテスト方式

3.2. ツールの選択

ユニットテストのテスト設計に新しいテストツールの適用を検討することとした．ユニットテストの自動化については様々なアプローチが研究されているが，本稿では，現場にて実用化するため，容易に試用でき，事前評価できることが求められることから，研修用に提供されている環境 (RAM は 2GB，CPU は Xeon E5520 2.26GHz，OS は CentOS) で CREST を用いることとした．CREST とはオープンソースとして公開されている C 言語向け concolic testing ツールである [9].

3.3. CREST 概要

CREST は Concrete Execution と Symbolic Execution を組み合わせた Concolic Testing を行うツールである．ここでは，まず Symbolic Exectution と Concolic Testing について説明する．

(1)Symbolic Execution:

Symbolic Execution は，プログラムの制御フローを解析する手段である．通常のプログラム実行は，変数に代入された値を使って実行するが，Symbolic Execution は変数を抽象解釈 (abstract interpretation) と呼ばれる方法で取扱い，各条件分岐の組合せを制約式として得る．得られた制約式を制約ソルバー (constraint solver) を用いて解く．

(2)Concolic Testing:

Concolic Testing は固定の具体的な値を使ったテストと Symbolic Execution を連携させて実行することにより，プログラムが持つすべての可達経路を網羅するテストデータを生成することを目標としている．経路探索は，Symbolic Execution により分岐点における分岐条件を求め，実行されなかった経路を選択するための入力値を制約ソルバーで求め，その具体的な入力値でプログラムを実行する，という流れである．

Symbolic Execution との違いは，すべてを疑似実行するのではなく，実コードでの実行とハイブリッドで行う点にある．この特徴を生かし，テスト対象の範囲を広く取り扱うことができるし，ランダム探索におけるスピードも改善されている．制約ソルバーでは解析的に解けない場合，乱数を用いたヒューリスティックな解を求めるが，そのランダム探索におけるスピードも改善されている．

4. CREST の網羅性確認

現行ユニットテスト方式の欠点を補うため CREST を導入するにあたり，我々のソースコードにおいて CREST により全可達経路のテストケースを生成できるかを確認する必要がある．そこで，いくつか関数を抽出し，CREST で確認を行った．確認を行う上で，問題点を評価するため，選んだ関数の持つロジックを抽象化したサンプル関数を作成して試用した．その理由は，色々なバリエーションを試すことと，問題点について外部からコメントをもらうとき，製品そのものが持つ情報を隠ぺいするためである．

サンプル関数の抽出方法は循環的複雑度（以下，CCN）を基に選択することとした．あるプロジェクト A で作成したソースコードの各関数の CCN の分布状況を確認すると大部分が 10 以下であることがわかった（図 2）．よって，サンプル関数として CCN が 2(小)，4(中)，8(大) の 3 ユニットを抽出して試用することとした．

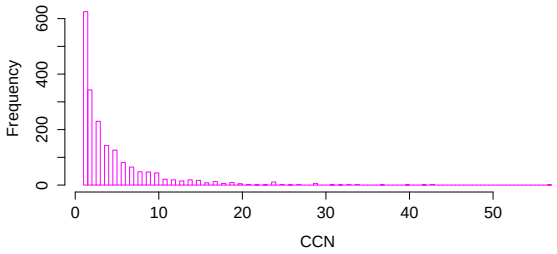

図 2. プロジェクト A で作成したソースコードの循環複雑度の分布

```
1 unsigned char ccn2( unsigned char a ){
2   unsigned char result ;
3   if ((a & 0x01U) == 0x01U){
4     result = 1U;
5   }else{
6     result = 0;
7   }
8       return ( result );
9 }
```

図 3. CCN=2 のサンプル関数

まず，CCN=2 のサンプル関数 (図 3) に CREST を試用した．探索オプションとして DFS(Depth-First Search) では，カバレッジ 100%のテストケースが生成されなかったが，その他の探索方法ではカバレッジ 100%のテストケースが生成された．DFS オプションの場合，例題のようなビット演算に対して，CREST が使っている制約ソルバー（yices）が対応していないことが原因である．韓国の SAMSUNG 社が携帯電話の OS に対して行った論文では，ビット演算のための機能追加を行ったことが書かれている [10]．ここでは，追加機能を用いず，DFS 以外の探索オプションとして CFG(Control-Flow Directed Search) を用いて解を求めた．

次に，CCN=4 のサンプル関数 (図 4) に CREST を試用した．CCN=2 のサンプル関数と同様に DFS 以外の探索方法でカバレッジが 100%のテストケースが生成されたが，引数 a が 5 行目で配列 c の領域外アクセスとな

```
 1   unsigned char ccn4( unsigned char a,
                         unsigned short b ){
 2     static const unsigned short c[2]=
                               { 510U, 514U };
 3     unsigned short d;
 4     unsigned char result;
 5     d = c[a];
 6     if ( b == 0U ){
 7       result = 2U ;
 8     }else if ( (d <= b) && (b <= 1000U) ){
 9       result = 1U ;
10    }else{
11      result = 0U ;
12    }
13    return ( result ) ;
14  }
```

図 4. CCN=4 のサンプル関数

```
1  int main( void ){
2    unsigned char a;
3    unsigned short b;
4    while ( 1 ){
5      CREST_unsigned_char( a ) ;
6      if ( a < 2U ){
7        break ;
8      }
9    }
7    CREST_unsigned_short( b );
8    printf( "%d %d\t\t %d\n", a, b, ccn4( a, b ) );
9  }
```

図 5. 入力値を制限する処理を追加したテストドライバ

```
 1 void func( void ){
 2  if ( Dat6 > 0U ){
 3      Dat6--;
 4      if ( Dat6 == 0U ){
 5        AAA[ Dat5 ]._dat2--;
 6        if ( AAA[ Dat5 ]._dat2 > 0U ){
 7          Dat6 = AAA[ Dat5 ]._dat1;
 8          if ( Dat1 == 0x01 ){
 9            Dat1 = 0x00;
10          }
12       } else {
13         Dat5++ ;
14         if ( Dat5 < 6 ){
15           Dat6 = AAA[ Dat5 ]._dat1 ;
16           Dat1 = AAA[ Dat5 ]._dat3 ;
17         } else {
18           Dat4++ ;
19           if ( Dat4 < Dat7 ){
20           } else {
21             if ( Dat3 == 0x01 ){
22               Dat4 = 0U ;
23             } else {
24               Dat2 = 0x00 ;
25             }
26           }
27         }
28       }
29     }
30    }
31  }
```

図 6. CCN=8 のサンプル関数

る値を生成していた．CREST はあくまでプログラムを実行して入力を求めているため，配列の領域外にアクセスして得られた値により分岐条件が成立すれば，それがテストデータになってしまう．

この問題を回避するために CREST が生成する a の値が 2 未満となるように条件文を追加した (図 5 の 4 行目，及び，6 行目から 9 行目)．その結果，適切なテストデータによりカバレッジが 100%となるテストケースが生成されることを確認した．

ここでテストドライバに print 文を使用しているが，CREST 以外の Symbolic Execution ツールでは，テストドライバで使用する print 文自身も経路探索の対象となり，ソースコードを与える必要が生じるが，CREST の場合は，その必要が無く使い勝手が良い．

最後に，CCN=8 のサンプル関数 (図 6) に CREST を試用した．CCN の増加だけでなく入力変数が 24 もある関数のため，何の工夫もせずに CREST によりテストケースを生成すると 14 分岐あるうちの 7 分岐まで網羅するテストケースは生成されたが，残り 7 分岐を網羅するテストケースは約 1 時間探索を繰り返しても生成されなかった．ここで，この CCN=8 のサンプル関数も変数 Dat5 が配列 AAA のインデックスとなっているため入力値を制限する処理を追加した．それにより生成されたテストケースによるカバレッジは増加したが 100%には至らなかった．そこで，可達経路を網羅する入力値の候補が多いために探索に時間がかかっていると考え，配列のインデックスとなる変数以外で取り得る値の範囲が決まっている変数に対しても入力値を制限する処理を追加した．その結果，約 1 分の探索時間でカバレッジが 100%となるテストケースが生成された．ここで行った

表 1. ユニットテストツールで生成したテストケースのカバレッジ

関数 No.	C0 カバレッジ	C1 カバレッジ	MC/DC カバレッジ	CCN
1	100.0%	100.0%	100.0%	4
2	100.0%	100.0%	100.0%	11
3	100.0%	100.0%	100.0%	12
4	100.0%	100.0%	100.0%	10
5*	75.0%	50.0%	33.0%	4
6	100.0%	100.0%	100.0%	1
7	100.0%	100.0%	100.0%	1
8	100.0%	100.0%	100.0%	2
9*	45.0%	50.0%	42.0%	8
10	100.0%	100.0%	100.0%	12
11*	97.0%	80.0%	50.0%	15
12*	52.0%	42.0%	20.0%	13
13	100.0%	100.0%	100.0%	1
14	100.0%	100.0%	100.0%	1
15	100.0%	100.0%	100.0%	1

表 2. CREST で生成したテストケース追加後のカバレッジ

関数 No.	C0 カバレッジ	C1 カバレッジ	MC/DC カバレッジ	CCN
1	100.0%	100.0%	100.0%	4
2	100.0%	100.0%	100.0%	11
3	100.0%	100.0%	100.0%	12
4	100.0%	100.0%	100.0%	10
5°	100.0%	100.0%	100.0%	4
6	100.0%	100.0%	100.0%	1
7	100.0%	100.0%	100.0%	1
8	100.0%	100.0%	100.0%	2
9°	100.0%	100.0%	100.0%	8
10	100.0%	100.0%	100.0%	12
11°	100.0%	100.0%	100.0%	15
12*	97.0%	95.0%	90.0%	13
13	100.0%	100.0%	100.0%	1
14	100.0%	100.0%	100.0%	1
15	100.0%	100.0%	100.0%	1

対策は，形式仕様で用いられている「事前条件」に相当すると考えられる．

以上のことから，CREST を我々のソースコードに適用できる目途付けを行うことができた．次章では，実在するプロジェクト A で作成したソースコードにおいて C0，C1，MC/DC カバレッジ 100%のテストケースが生成できなかった関数に対して，実際に CREST を適用し現行ツールの問題を解決できるかを確認する．

5. 現行ユニットへの CREST 適用結果

ここでは，プロジェクト A で作成した実際のソースコードに対して，CREST が現行ツールより優れているか否かを比較評価する．

まず，プロジェクト A のライブラリから無作為に C ソースファイルを抽出し，そこに含まれる関数に対して，現行ユニットテストツールによりテストケースを生成させた．その結果を表 1 に示す．対象とするソースファイルに含まれる 15 関数の内，* 印の 4 つの関数において C0，C1，MC/DC カバレッジ 100%に未達であった．

それらの関数に対して CREST によりテストケースを生成し直した結果，4 つのうち ° 印の 3 つの関数において C0，C1，MC/DC カバレッジ 100%となるテストケースが生成された (表 2)．残り 1 つの関数においてカバレッジが 100%とならなかった理由は 2 つあり，1 つは防衛的プログラミングによりメモリ破壊が発生しない限り通らない経路である．もう 1 つは図 7 のように 1 行目の 2 つ目の条件式が成立しない場合は 3 行目の 2 つ目の条件式が必ず成立するため無駄な処理となっていたためであった．

```
1 if ( (a == FALSE) && (b == TRUE) ){
2 ……
3 }else if ( (a == FALSE) && (b != TRUE) ){
4 ……
5 }
6 else {
7 }
```

図 7. 未達経路 の例

以上のことから，現行のユニットテストツールで全可達経路のテストケースを生成できなかったユニットに対して，この実験では CCN の大小に関わらず CREST により全可達経路のテストケースを生成することが確認できた．

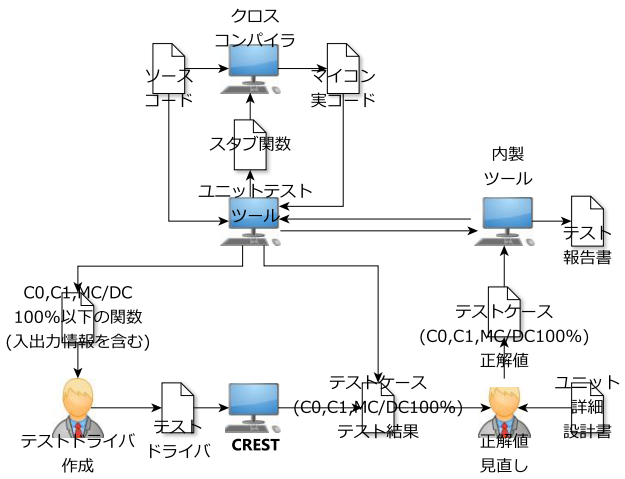

図 8. CREST 導入後のユニットテスト方式

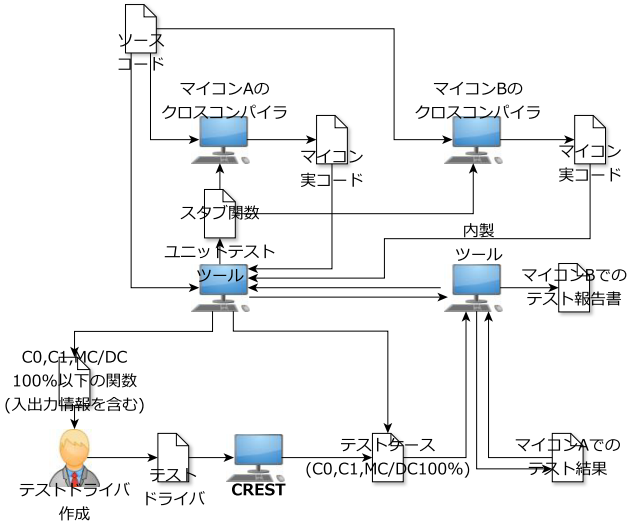

図 9. CREST 導入後のデグレードテスト方式

6. CREST によるユニットテスト方式の改善

本章では，2 章で示した現行のユニットテスト方式に対して，どのように CREST を導入するかを述べる．

まず，現状の方式ではユニットテストツールが出力するテストケースに対して，カバレッジ 100%となるテストケースと正解値を設計書を基に追加する．CREST 導入により，設計書を基にテストケースを追加する作業が，ユニットテストツールが生成した関数の入出力情報から CREST 用にテストドライバを作成する作業に置き換わる．この作業は，設計情報を知らなくてもできる作業のため，設計者のスキルがある人以外でも担当することができる．よって，設計情報を知らないとできない作業は正解値を設計書を基に見直す作業だけとなる (図 8)．これにより，ユニットテストでかかる工数は大幅に削減されると考えられる．

また，自職場ではマイコン変更，及び，コンパイラ変更を伴う派生開発も多々あり，その際に，実施するデグレードテストには，CREST の適用が更に有効である．その方式を図 9 に示す．現状，派生元のプロジェクトではユニットテストで実施可能なテストケースを実機を用いた結合テストに含めて実施していたため，デグレードテストにおいてユニットテストを実施する場合，一からテストケースの作成を余儀無くされる．しかし，CREST を導入すると可達経路を通るテストケースを得られるため，派生元のマイコン実コードと得られたテストケースを使いテスト結果を生成し，それをデグレードテストの期待値とすることで，ほとんど人手を必要としなくなる．但し，4 章で説明したようにただ入力を与えるだけでは可達経路を通るテストケースを得られない場合があり，入力値を制限する処理の追加は必要となる．また，CREST の特性やパラメータの理解と，テスト対象ユニットが持つドメイン固有のレジスタ操作などの特殊性を理解するスキルも求められる．

7. おわりに

C 言語向け Concolic Testing ツールである CREST が我々のソフトウェアに適用可能であることをサンプル関数，及び，実際の製品ソフトウェアを使って確認できた．これにより，ユニットテストにおいて課題の一つであったテストケース作成の効率化に CREST を用いることで解決の目途がついた．また，ユニットテストにおけるもう一つの課題である正解値の作成も派生開発におけるデグレードテストでは解決可能であることが確認できた．

当初，オープン系のツール CREST は，商用ツールと比べ入門向けの情報やノウハウが少なくとても困惑した．特に，動作環境を構築するのは，CREST を構成している様々なオープン系ツールの依存関係があり難しい．実際に CREST が動作する環境があれば，その環境下で試行的に使いながら問題を解決して行くのは，それほど難

しくなく，短期間で進めることができた．

今後の課題として，デグレードテストは直ぐにでも改善すべき方式であるため，現場の担当者が容易に使えるような環境の整備と保守方式を検討する．また，CREST でテストケースを生成する上で，変数の各ビットが ON/OFF のような意味を持つ入力のテストケース生成は変数のサイズが大きくなるにつれて長くなる生成時間の短縮が課題である．

最後に，この研究は，デンソー技研センターの 2014 年度ソフトウェア工学ハイタレント研修の中で行った．講師の先生と，研究の機会を与えていただいた職場に感謝する．

参考文献

[1] 大須賀竜治，“ISO26262 の日本自動車業界での活動と今後について (< 連載 > ISO 26262-自動車業界における機能安全に対する取り組み-),” 品質，vol.43，no.2，pp.212–217，2013.

[2] 梅村晃広，“SAT ソルバ・SMT ソルバの技術と応用,” コンピュータ ソフトウェア，vol.27，no.3，pp.24–35，2010.

[3] K. Sen, D. Marinov, and G. Agha, CUTE: a concolic unit testing engine for C, vol.30, ACM, 2005.

[4] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineeringIEEE Computer Society, pp.443–446 2008.

[5] J. Foster, “Symbolic execution,” http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec13-SymExec.pdf.

[6] W. Visser, C.S. Psreanu, and R. Pelánek, “Test input generation for java containers using state matching,” Proceedings of the 2006 international symposium on Software testing and analysisACM, pp.37–48 2006.

[7] C. Cadar, D. Dunbar, and D.R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.,” OSDI, vol.8, pp.209–224, 2008.

[8] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineeringIEEE Computer Society, pp.443–446 2008.

[9] CREST, “Concolic test generation tool for c,” http://jburnim.github.io/crest/.

[10] Y. Kim, M. Kim, Y.J. Kim, and Y. Jang, “Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee,” Software Engineering (ICSE), 2012 34th International Conference onIEEE, pp.1143–1152 2012.