Concolic Testing ツール CREST 利用方法の提案

― リグレッションテストへの適用に向けて –

Suggestion of utilization for Concolic Testing tool CREST

- Application to regression testing –

主査　　：奥村 有紀子　（有限会社デバッグ工学研究所）
副主査　：秋山 浩一　　（富士ゼロックス株式会社）
副主査　：堀田 文明　　（有限会社デバッグ工学研究所）
研究員　：村上　仁　　（株式会社ニコン）

研究概要
現在，Concolic Testing は世界的に利用技術の研究が進められている．しかし，それぞれの分野ごとの報告となり，まとまった一連の利用解説がないために[1][2][3]，実業務に利用するには利用法を調査して知識を積むことが必要となる．本研究では，Concolic Testing ツールの CREST を用いて，リグレッションテストへの利用方法を体系的にまとめることを目指した．そして，仕様・論理についてリグレッションテストに適用できることを検証し，その方法を提案した．

Abstract
There have been studies about technologies used to conduct concolic testing worldwide. But there is no document that thoroughly explains the process, thus have to gain knowledge through experience in order to make use in real-world testing. This study aimed to see if the concolic testing tool CREST can be applied to regression tests, and to systematize the usage. In process it shows how it can be applied to regression tests to check the specifications and theory behind the tested software.

1.　はじめに

1.1　研究の背景と課題

現在,Concolic Testing は世界的に利用技術の研究が進められている[1][2][3][4][5]. プログラムを自動解析してテストを自動実行するテスト技術である Concolic Testing は,従来のソフトウェア検証を変えてしまう大きな可能性を持っていると考えられる[5].そのため，この分野における利用技術を早期に確立することは企業や産業にとっては重要度が高い．しかし，先行研究やツール開発者の利用方法に関する説明はそれぞれの分野ごとの報告となり，まとまった一連の利用解説がない．また，詳細な利用方法は企業のノウハウとして非公開になっている例もある．そのため，製品のテストに利用するには利用方法を個々に調査して，知識を積むことが必要となる．一方，研究メンバーの組織では，テスト不足によるバグの検出漏れがあり，効果的なリグレッションテストが実施できていないという課題がある．しかし，リグレッションテストについては，調査した範囲の先行研究[1][2][3][4][5]で事例を見つけられなかった．そこで，Concolic Testing を製品に利用するための調査テーマとして，Concolic Testing ツールの一つである CREST を用いて，リグレッションテストへの利用方法を体系的にまとめることを考えた.

1.2　研究の目的

本研究では，Concolic Testing のリグレッションテストへの利用方法を体系的にまとめることを目的とする．

2. Concolic Testing とは

2.1 Concolic Testing 概要

「Concolic」とは Concrete(具体値)と Symbolic(記号)の合成造語である．Concolic Testing は，Symbolic Execution(記号実行)と Concrete Execution(具体値実行)により全ての制御パスを通るテストケースを自動生成し，コンパイルした実プログラムにて実行する自動テスト技術の一種である[1][5]． 1970 年代に理論が提唱され，コンピュータの発達により 2005 年頃からツールが商品化され始めた．2008 年頃からオープン環境での利用が可能になり，2012 年頃から実用面での試行が始まっている[5]．

2.2 Concolic Testing の原理

Symbolic Execution(記号実行)は，変数を抽象解釈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プログラムの制御フローを解析する手法である．そして，得られた制約式に対して変数の適合する範囲値を求める機能である制約ソルバーを用いて解いていく．制約式の解を見つけ出す Symbolic Execution(記号実行)ができなくなった場合には，適当な値を用いてプログラムを実行する Concrete Execution(具体値実行)によって通る制御パスを確認することができる[5]．

2.3 Concolic Testing ツール CREST

本研究では,Concolic Testing ツールの中から C 言語用の CREST について検証する.CREST は，2008 年にリリースされ，オープンソースとして提供されている．テスト対象のソースコード自体を扱うので，解析のために別途モデルを作成する必要がない．また，生成されたテスト入力と通過した制御パスに関する情報や実行結果についても記録が残るため検証しやすいという特徴を持っているので，研究対象のツールとして採用した．

CREST は，C 言語から曖昧性を排除した CIL(C Intermediate Language)と呼ばれる中間言語に変換後，CRESTC コマンドで中間言語に展開されたソースコードを構造解析して可達な制御パスを自動検出する．コンパイラには GCC を用いる．そして，run_crest コマンドで制約ソルバーYices を使って変数の制約の解を求め，変数の具体値を発生させて可達パスをテストする[5]．

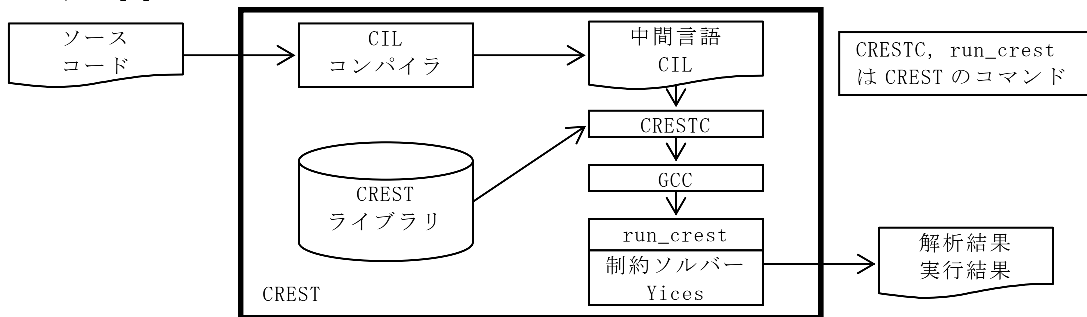

図 1. CREST イメージ図

2.4 CREST の制御パス解析方法

CREST の制御パス解析方法の一つに本研究で採用した深さ優先探索(depth-first search)がある(以下，dfs と呼ぶ)．dfs は，最初のノードからゴールが見つかるか，解けないノードに行き着くまで深く掘り下げて探索を行い，ゴールノードが見つからなかったらまだ通っていないノードまで引き返して，可能な限り探索を続けるアルゴリズムである．この方法を使って CREST では，可達パスを通るデータを見つけて，実行結果を出力する．

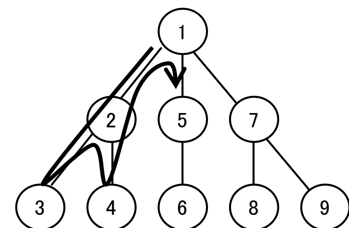

図 2. 深さ優先探索イメージ

2.5 対象プログラムの CREST 対応

対象プログラムを CREST で使用するには，プログラムの Symbolic 入力生成のために，CREST 実行時に制約ソルバーが扱う Symbolic 変数を宣言する必要がある．(付録 4(1))

3. リグレッションテストへ利用するための道筋

CREST をリグレッションテストに利用するために，以下のような手順で検証する．

(1)旧版における CREST 検証の適正さを確保する

①パスの網羅性を確認する

CREST が解析した制御パスの確認方法の確立と，解析・テスト実行のパスの網羅度，未達パスの解析方法を確立する．

②仕様との一致性を確保する

CREST はプログラム構造に対するテストであり，仕様との一致性の検証が必要．

③バグの摘出方法を確認する

実行結果だけからは判断できないバグの摘出方法を確認する．

(2)旧版と新版のテスト結果比較方法を確立する

新版のデグレード検証には，旧版の入出力データと新版の入出力データを比較する．そのテスト結果の比較方法を確立する．

デグレードを検出するために初めて CREST を適用する場合，変更前プログラム(旧版)と変更後プログラム(新版)の両方を検証の対象として実行する必要がある．このため，まず旧版における CREST 検証の適正さを確保し，次に新版のテスト結果と比較しデグレードの有無を確認する．なお，CREST で実行可能なテストは仕様・論理の検証であり，それ以外のシステムの振る舞い(マルチセッションの同期など)は，別の手段が必要になる．後者については，本研究では対象外とする．

4. 旧版における CREST 検証の適正さ確保

4.1 検証用のプログラム

本研究では，検証用のプログラムとして三角形形状判定のプログラム(付録 3)を作成した．仕様は以下である．

(1)三角形を構成する 3 辺の長さを入力する．
(2)各辺の入力値は正の整数でなければならない．
(3)正の整数以外の値が入力されると，入力データ無効メッセージと 10 を出力する．
(4)各辺の長さの関係を計算して，三角形を形成しないと判定すると，三角形が形成できない旨のメッセージと，10 を出力する．
(5)正三角形と判定すると，40 と出力する．
(6)二等辺三角形と判定すると，30 と出力する．
(7)不等辺三角形と判定すると，20 と出力する．

プログラムの内容は付録 4 に掲載した通りである．このプログラムには，CREST が検証に必要な結果を出力するための工夫として，以下の仕掛けのいずれかを施す(付録 4(2))．

①CREST が変数値を出力するために CREST のプログラムにパッチを充てる．
②CREST が変数に対して自動生成する入力値と結果を収集するために印刷(printf)文を挿入する．

4.2 パスの網羅性確保

まず，CREST が実行した制御パスを解析する．CREST が解析実行時に出力した中間言語の構造情報を CREST の出力する Configuration（CFG）情報を使って作成する(付録 4(7))．次に，run_crest コマンド実行時の生成された入力値を手動で抽出する(付録 4(5))．そして，中間言語の構造情報と，CREST の dfs 指定によるテスト結果(付録 4(4)run_crest コマンドによるテスト実行結果)と生成された入力値からブランチとノードの網羅度を把握する．未達パスがなければブランチをカバーできているので OK とする．この例では 11 回の繰り返しで中間言語展開後の制御パス（厳密には CREST の解析から可達なパス）が網羅されたということである．

本論文では言及しないが，未達パスがある場合には中間言語の構造情報と入力・結果の

組み合わせ，run_crest コマンドの出力であるカバレッジファイルのデータを使って調査する必要がある．

4.3　仕様との一致性確保

ブラックボックステストにおいて，プログラム構造情報は不明の場合が多い．例えば，三角形形状判定プログラムにおいて処理方法は推測可能だが，推測したようにプログラムが実装されているとは限らない．そこで，先行研究[4]を参考に，仕様のデシジョンテーブル(以下，DT と記述)と CREST によるプログラムのテスト実行結果から得られる入力値と出力値の組み合わせを表記した DT を作成して,原因(入力値)と結果(出力値)の関係から処理を検証した．先行研究は仕様から DT を自動生成し，CREST テスト結果との一致性も自動検証することを狙っているが，仕様表現とプログラム論理を一定の型に当てはめないと適用できないことから，我々の DT は手動で作成することにした．

4.1 の三角形形状判定プログラムの仕様から DT を作成した．

表 1．三角形形状判定プログラム:仕様の DT

規則				1	2	3	4	5	6	7
条件部	辺長	A ＞ 0	Y/N	Y	Y	Y	N			Y
		B ＞ 0	Y/N	Y	Y	Y		N		Y
		C ＞ 0	Y/N	Y	Y	Y			N	Y
	三角形形成		Y/N	Y	Y	Y	N	N	N	N
	三辺等しい		Y/N	Y		N				
	二辺等しい		Y/N		Y	N				
動作部	正三角形(40)			X						
	二等辺三角形(30)				X					
	不等辺三角形(20)					X				
	エラー(10)		不当入力				X	X	X	
			三角形非形成							X

プログラムを run_crest コマンドでテスト実行し，実行結果から Iteration ごとの入力値を手動で組合せて(付録 4(6))，CREST 実行結果の DT に表したのが表 2 である．

表 2．三角形形状判定プログラム:CREST 実行結果の DT

テストケース			1	2	3	4	5	6	7	8	9	10	11
原因	辺長	A	0	1	1	1	1	2	1	2	3	4	2
		B	0	0	1	1	1	1	2	1	2	3	2
		C	0	0	0	1	2	1	1	2	2	2	1
結果	Invalid(10)		X	X	X								
	三角形非形成(10)						X	X	X				
	正三角形(40)					X							
	二等辺三角形(30)									X	X		X
	不等辺三角形(20)											X	

仕様の DT の規則に対応したテストケースが実施されているか表 3 にてチェックすると，仕様の各規則に対応したテストケースが実施されている．このような形で外部仕様とプログラム動作の一致性を評価する．ただし，入出力に差異がある場合は一致していないと言えるが，差異がない場合に，途中の処理の適正さなどを詳細に確認するには，CREST が生成した変数値の分析や，バグの存在有無を確認する必要がある(4.4.2　assert 文を挿入することにより検知できるバグの例参照)．

なお，表 2 のテストケース 1，2，3 は付録 4(1)のプログラムの 29 行目の入力データ誤りチェックに対応している.CRESTC コマンドによる中間言語展開後の制御パスは付録 4(7)のようになる．dfs で解析するとテストケース 1 で A が不正値，テストケース 2 で B が不正値(A は正常),テストケース 3 で C が不正値(A と B は正常)と順にチェックしていることがわかる．また，本例題について，各変数の境界値(0，1)テストも実行できている．

表 3. 仕様と CREST 実行結果の比較

仕様の規則	テストケース	対応状況
1	4	対応
2	8，9，11	C=A，B=C，A=B の 3 ケースがテストされている.
3	10	対応
4	1	対応
5	2	対応
6	3	対応
7	5，6，7	A＋B≦C，A+C≦B，B+C≦A の 3 ケースがテストされている

4.4 バグの摘出検証

CREST によってどのようなバグを摘出できるかいくつかのバグを例に検証する.

4.4.1 CREST を実行すればわかるバグの例

付録 5 のように，三角形として成立するかのチェックを if ((iSide1 + iSide2)< iSide3 || (iSide2 + iSide3)< iSide1 || (iSide1 + iSide3)< iSide2)とした. この時の CREST 実行結果は表 4 である. テストケース 8，9，10 は本来非形成と判定されなければならないが，二等辺三角形と判定されており，バグがあることがわかる.

このように入力値と結果を比較することによりわかるバグは，CREST を実行することで摘出できる.

表 4. バグ有三角形形状判定プログラム 1:CREST 実行結果の DT

テストケース			1	2	3	4	5	6	7	8	9	10	11
原因	辺長	A	0	1	1	1	1	3	1	2	3	1	2
		B	0	0	1	1	1	1	3	1	2	2	2
		C	0	0	0	1	3	1	1	1	1	1	1
結果	Invalid(10)		X	X	X								
	三角形非形成(10)						X	X	X				
	正三角形(40)					X							
	二等辺三角形(30)									X		X	X
	不等辺三角形(20)										X		

4.4.2 assert 文を挿入することにより検知できるバグの例

実行結果は一見正常に見えても，途中の中間結果に誤りが出る時や，データによっては異常を起こす時は，バグを検知できないことがある. このような場合は，assert 文による判定のチェックや,論理処理に影響しない if 文などの追加によりバグを検知する必要がある(付録 6). これらの検証は通常のプログラムデバッグと同じであるが，CREST の入力変数値自動生成により，より簡単に実行できる方法がある. 例えば，0 による除算(0 割)は一般的には分母が 0 になるケースを作り出さなければならないが， CREST が 0 になる条件を算出しない時は，付録 7 のように分母!= 0 という assert 文を挿入すると，割り算の文の前に分母の変数に様々な加工があったとしても，CREST は 0 となる条件が成立可能かを検証することが可能となる. 0 値ポインター，領域外アクセスなどのチェックも通常のデバッグに CREST の入力値自動生成を利用してほぼ同様に行うことができる. その他，ループ処理は CREST の解析では最短パスを通るので，ループを繰り返したい場合は assert 文や if 文挿入などを工夫する必要がある.

旧版における CREST 検証の適正さを評価するために，パスの網羅性，仕様とプログラムの一致性，バグの摘出方法を検証した. パスの網羅性は，中間言語の構造情報と，CREST のテスト結果と生成された入力値から把握できる.仕様とプログラムの一致性については，仕様の DT と CREST 実行結果の DT を比較することにより検証した. 実行結果に差異があると一致していないと言えるが，差異がない場合，一概には一致するとは言えない.例えば，入力値と結果に不整合の出ないバグについては摘出できていないので，バグの摘出も行う

必要があり，摘出方法の例を示した．以上のように，パスの網羅性の確認，仕様とプログラムの一致性の確認，バグの摘出を行えば，旧版における CREST 検証の適正さを確保できる．

5.　旧版と新版の比較

5.1　リグレッションテストの検証

リグレッションテストについては，新旧プログラムの CREST 実行結果の DT を比較することにより，デグレードの有無を判断できるか検証する．旧版の適正さが確保できている前提で(4 章),仕様変更がなく旧仕様のままの箇所は，仕様・論理において

(1)　入力値：同じ，結果：同じ　→　OK(デグレードなし)
(2)　入力値：同じ，結果：違う　→　NG(デグレードあり)
(3)　入力値：違う　→　調査必要(仕様外に改修された可能性あり)

と判断できる．仕様追加/変更のない箇所は上記の方法で判断し，追加/変更箇所は 4 章と同様の方法で確認する．

5.2　新版のデグレード確認

付録 4 の三角形形状判定プログラムに以下の辺長の上限を仕様追加(付録 8)して実施した．

・各辺の長さが 10 より大きいとエラーとし，5 と出力する．

この仕様追加を反映した仕様の DT が表 5 である．

表 5．新版の三角形形状判定プログラム:仕様の DT

規則				1	2	3	4	5	6	7	8	9	10
条件部	辺長	A ＞ 0	Y/N	Y	Y	Y	N			Y	Y	Y	Y
		B ＞ 0	Y/N	Y	Y	Y		N		Y	Y	Y	Y
		C ＞ 0	Y/N	Y	Y	Y			N	Y	Y	Y	Y
		A ＜ 11	Y/N	Y	Y	Y					N		
		B ＜ 11	Y/N	Y	Y	Y						N	
		C ＜ 11	Y/N	Y	Y	Y							N
	三角形形成		Y/N	Y	Y	Y	N	N	N	N			
	三辺等しい		Y/N	Y		N							
	二辺等しい		Y/N		Y	N							
動作部	正三角形(40)			X									
	二等辺三角形(30)				X								
	不等辺三角形(20)					X							
	エラー(10)		不当入力				X	X	X				
			三角形非形成							X			
	エラー(5)		辺長オーバー								N	N	N

仕様変更に基づき変更したプログラムに対し，CREST によるテスト実行結果である Iteration の入力値(付録 8(4))と結果(付録 8(3))から CREST 実行結果の DT に表したのが表 6 である．

表 6．新版の三角形形状判定プログラム:CREST 実行結果の DT

テストケース			1	2	3	4	5	6	7	8	9	10	11	12	13	14
原因	辺長	A	0	1	1	1	11	1	1	1	2	1	2	3	4	2
		B	0	0	1	1	1	11	1	1	1	2	1	2	3	2
		C	0	0	0	1	1	1	11	2	1	1	2	2	2	1
結果	Invalid		X	X	X											
	三角形非形成									X	X	X				
	辺長オーバー						X	X	X							
	正三角形					X										
	二等辺三角形												X	X		X
	不等辺三角形														X	

仕様変更箇所の仕様と CREST 実行結果の比較は表 7 であり，仕様変更がプログラム動作に反映されていることがわかる.

表 7. 仕様と CREST 実行結果の比較

仕様の規則	テストケース	対応状況
8	5	対応
9	6	対応
10	7	対応

仕様から境界値は 10 と 11 と特定できるが, 10 についてはテストされていないことがわかる. しかし, 11 がテストデータとして選定されていることから, エラー側境界値として 11 を CREST が認識していることが推定される. このため, プログラムが 10 を正常側に区分していることが推定される. さらに厳密に検証したい場合はデバッグモードで assert 文の挿入などにより確認することが必要である.

新版の仕様変更部分(表 6 のテストケース 5,6,7)を除いて旧版の CREST 実行結果の DT(表 2)と比較すると，テストケースの原因(入力値)と結果は同じである. 以上より，追加箇所の仕様との一致性と，外部仕様からの処理においてデグレードはないと判断できる.

5.3　仕様変更後デグレードが発生した場合

5.2 の辺長の上限追加に加えて, 三角形であることの判定文にバグを入れた(付録 9 参照). この時の CREST 実行結果は表 8 である.

表 8. 仕様追加とバグ有三角形形状判定プログラム:CREST 実行結果の DT

テストケース			1	2	3	4	5	6	7	8	9	10	11	12	13	14
原因	辺長	A	0	1	1	1	11	1	1	1	2	1	2	3	4	2
		B	0	0	1	1	1	11	1	1	1	3	1	2	3	2
		C	0	0	0	1	1	1	11	2	1	1	2	2	2	1
結果	Invalid		X	X	X											
	三角形非形成									X	X	X				
	辺長オーバー						X	X	X							
	正三角形					X										
	二等辺三角形												X	X		X
	不等辺三角形														X	

表 8 を旧版の表 2 と比較すると，テストケース 5, 6, 7 が追加されており，テストケース 10 の入力値が変わっていることがわかる. 追加箇所については，5.2 で検証済である. 10 は入力値が違うので，調査が必要となる. 非形成のメッセージが出力されるのは三角形であることをチェックする箇所なので，付録 6 の該当箇所を確認する. 本来，(iSide1 + iSide3)<= iSide2 とあるべき箇所が，(iSide1 + iSide3)< iSide2 と判定文が誤っており，デグレードが発生していることがわかる.

このように，仕様変更前後の CREST 実行結果の DT を比較するとプログラムの論理処理の変化が分かる. 旧版の論理処理が正しければ，仕様変更箇所以外について相違点がなければ，外部仕様の論理処理においてはデグレードがないと判断する. 本来仕様変更箇所がない箇所に相違があった場合には誤った変更，すなわちデグレードが発生したと推定できる.

6. まとめ

パスの網羅性の確認，仕様の一致性確保，バグの抽出により旧版における CREST 検証の適正さ確保し，旧版と新版のテスト結果比較方法を検証することにより，CREST で仕様・論理におけるリグレッションテストに適用できることが検証できた.

さらに，CREST では実行可能な可達パスに対して自動テストが実行されるため，この範囲でも従来手法のテストと置き換えられる部分があると期待できる.

このように，CREST の実業務へ展開するにはすべてを置き換えるのではなく，CREST 適用可能範囲については CREST を利用し(本研究では CREST の実行結果を用いた DT)，その他を従来手法のテストで実施するというように，CREST によるテストと従来手法のテストを併用することで，現状よりも効果的なテストの実施が期待できる．

しかし，今後の検証が必要な課題もある．実行結果の分析だけではわからないバグ，すなわち，assert 文や if 文を工夫しなければならないバグの検出方法や変数生成については今回の研究では十分に検証できていない．さらに，CREST を使用する上で以下のような問題もあった．今回の研究においてデグレード検証として，三角形の形状判定プログラムに辺長の上限チェックを仕様追加したが,当初は直角三角形判定を追加する予定であった．ところが，CREST の制約ソルバーである Yices がべき乗とルート演算に対応しておらず，$A^2+B^2=C^2$ を解析できなかったため，検証を断念した．このように，ツール自体にもまだ解決されていない点があり，複雑なプログラムのテストに利用するには制約ソルバー自体の仕様を確認することが必要である．

以上のように実業務へ展開するには課題はあるものの，本研究により Concolic Testing の体系的利用方法をまとめる目途がついた．

7. 今後の展開

本研究では，三角形形状判定プログラムを作成して検証したが，実際の製品は構造が複雑で，ステップ数も多く複雑である．実用にはさらに CREST の使い方を調査しクリアにする必要がある．

① 試行などによって CREST が非対応である型の変換方法を調査する
② 大規模プログラムや複数ファイルにまたがるプログラムの扱い方を検証する[5]
③ 今回検証していないバグの見つけ方を検証する
　バグの例)永久ループ，デッドコード，NULL ポインタアクセスなど
④ Concolic Testing の体系的利用方法をまとめたガイドを作成する

上記の検証を今後の課題とする．

8. 参考文献

[1] K. Sen, D. Marinov, and G. Agha：‘CUTE: A concolic unit testing engine for C’, 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’05), ACM(2005)
[2] Moonzoo Kim, Yunho Kim,Gregg Rothermel , A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation, 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 978-0-7695-4670-4/12, 340-349, 2012
[3] Moonzoo Kim, Yunho Kim, Yoonkyu Jang, Industrial Application of Concolic Testing on Embedded Software: Case Studies, 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 978-0-7695-4670-4/12, 390-399, 2012
[4] 植月 啓次，ソフトウェアの実装情報に基づく決定表を活用した論理検証手法，ソフトウェア・シンポジウム 2013, 2013
[5] 松尾谷 徹，Concolic testing と背景技術～テスト技法の新動向～，ソフトウェアエンジニアリングシンポジウム 2013, 2013