社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－EVA－12（2）
2005／3／23

テストケースの適応的生成による頑健性テストの高速化

福永拓男[†]　中西恒夫[††]　福田　晃[††]

異常入力に対して脆弱なソフトウェアは少なくない．本稿では，Carnegie Mellon 大で開発された，異常入力に対するソフトウェアの頑健性テストを行うツール Ballista に改良を施し，その高速化を施す．Ballista のテスト生成過程に，テストケース生成順序を変更することによってランダムテストの高速化を図る手法 Adaptive Random Test(ART) by Localization 法を導入したものと，本稿提案のテストケース生成手法を導入したものを比較・評価を行う．POSIX 関数に対する評価の結果，本稿提案方式は ART by Localization 法よりもテストケース生成のオーバーヘッドが小さく，最大で約 50% のテスト時間短縮が可能であることを確認した．

Acceleration of Robustness Testing Using Adaptive Generation of Test Cases

Takuo Fukunaga,[†] Tsuneo Nakanishi[††] and Akira Fukuda[††]

Not a few softwares are crashed by abnormal inputs. In this paper, we improve Ballista, a robustness testing tool for abnormal inputs developed by Carnegie Mellon University, to shorten time for robustness test. We combined ART by Localization, a technique for accelerating random test by changing the order of test case application, and our proposing test case generation method with Ballista. As a result of evaluation by POSIX function robustness testing, our proposing test generation can shorten robustness testing time by Ballista to 50% at maximum with less overhead of test case generation than ART by Localization.

1. はじめに

現在，組込みシステムは医療機器や金融システム，工場の制御システムなどさまざまな分野で応用されている．それらの組込みシステムの中には社会において重要な役割を担っているものが多くある．それゆえ，組込みソフトウェアの誤動作は時として人命に関わる可能性がある．例えばアメリカ，テキサス州の病院において，加速器治療装置 (カナダ製 Therac25) の組込みソフトウェアのバグにより患者に電子線を直接照射し，2 名が死亡した事故があった[1)]．

ソフトウェアのエラーの中にはごく稀に発生する例外的・異常な入力に起因するものがある．ここでの例外的または異常な入力とは，サービス提供のための事前条件を満たさない入力や開発時に予期していなかった入力を意味する．このような例外的・異常な入力に起因するエラーを発見するためには頑健性テストを行う必要がある．頑健性テストには限られたテスト時間内でエラーを効率的に発見することが望まれる．

頑健性テストのひとつとして，Ballista テスト[2)]がある．Ballista テストは Carnegie Mellon 大学の P.Koopman らによって開発されたブラックボックステスト手法であり，ソフトウェアモジュールの API レベルでのテストを行う．しかし，既存の Ballista ツールのテストケース生成順序には改善の余地があり，テスト時間を短縮することが可能である．

本研究はテストケースを効率的に生成することで，Ballista テストによるエラー発見の高速化を目的とする．ここで，本研究では高速化という語を，従来手法と比較してより早い時間で最初のエラーを見つけることと定義する．高速化するために以下の二つの手法を実装し，効果を検証した．

一つ目の手法は Adaptive Random Testing(ART) である．ART の基本的な考え方はそれ以前に実行されたテストケースを考慮した上で，次のテストケースを定義域空間上において可能な限り均等に分布するように生成することである．また，二つ目の手法は我々が提案する「シャッフル法」である．シャッフル法は一度テストした値はそれ以降使用しないことを基本的な考え方とする．特定の 1 引数のみが寄与して引き起こ

† 九州大学 大学院 システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†† 九州大学 大学院 システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University

されるエラーの分布は一般的にも多く存在し[3]，シャッフル法はこのような分布に対して効果がある．また，ART と比較して実装が容易であり，テストケース生成のオーバーヘッドが小さいことも利点である．

本研究の以降の章の構成は以下の通りである．まず、第 2 章では関連研究として，Ballista テストと ART を紹介する．第 3 章では Ballista テストを実行し，POSIX 関数に現れるエラーの分布傾向について分析を行う．第 4 章ではシャッフル法を提案し，第 5 章で ART とシャッフル法に関する実験と考察を行う．最後に第 6 章で二つの高速化手法についてまとめて，今後の課題について述べる．

2. 関 連 研 究

2.1 Ballista テスト

Ballista テストは，Carnegie Mellon 大学の P.Koopman らによって開発された，ソフトウェアモジュールの API レベルでのテスト手法である．Ballista テストを用いることで，さまざまなソフトウェアモジュールの頑健性に対するブラックボックステストを行うことができる．Ballista テストでは，各データ型ごとにテスト値リストを用意する．各対象モジュールのテストケースを生成する際にはそのモジュールの各仮引数の型について定義されているテスト値リストの中からテスト値を選択する．このようにテストケースの生成方法が極めて単純であるため，テストケース生成の自動化は容易である．また，Ballista ツールは移植性が高く，広範なオペレーティングシステム上で実行可能である．これまでに複数の UNIX 系 OS 間で頑健性を比較している．

2.2 Adaptive Random Testing (ART)

ART はランダムテストと比較して，より短時間でエラーを発見できるように改善した手法である．ART ではテスト対象モジュールの入力パラメータの定義域空間上でのテストケースのユークリッド距離を計算することが多い．順次生成するテストケースを既に生成したテストケースから一定の距離以上となるようにし，テストケースを定義域空間上に均等に分散させる．これにより定義域空間上に連続的に分布するバグをより少ないテストケースで検出することを図る．従来までに提案されてきた ART として代表的なものを以下に挙げる．

Distance-based ART(D-ART) では，定義域空間全体からテストケース候補をある特定の個数，ランダムに生成する．それらの候補ひとつひとつに対し，既に生成した全テストケースとの間の距離を求め，定義域空間上で均等に分散するように次に実行するテストケースを選ぶ．

Restriction Random Testing(RRT) では，既に生成した全てのテストケースの周囲のある一定距離を排除領域とする．そしてテストケース候補を順次，ランダムに生成し，全ての排除領域の外にある候補が次に実行するテストケースとして選ばれる．

ART by Random Partition では，テストケース生成領域 (最初は全定義域空間) の中からテストケースをランダムに生成して，その点を基に生成領域を異なる面積の領域に分割する．分割された領域のうち，面積が最大のものを次のテストケース生成領域とする．同様にしてテストケース生成，領域分割をテストが終了するまで繰り返す．

しかし、以上で挙げた ART に対してはいくつかの欠点が指摘されている．D-ART や RRT においてはそれ以前に生成した全てのテストケースとの間の距離を計算しなければならないため，テストが進むに連れ，距離計算の手間は膨大になってしまう．ART by Random Partition においては，距離計算こそ行わないが，テストケースが定義域空間の中に十分均等に分散しない場合があることが分かっている．

T.Y.Chen らによる ART by Localization[4] はこのような欠点を補うものである．この手法は領域分割を行う際に D-ART と RRT を導入したものである．距離計算を大幅に簡略化することができ，かつ ART by Random Partition の欠点を解決している．ART by Localization には D-ART を利用する D-ART by Localization と RRT を利用する RRT by Localization の二つがある．

3. Ballista テストの実行

我々は Linux2.6.9 で実装されている POSIX 関数のうち，144 の関数に対して，各引数に用意されている全てのテスト値の組合せについて Ballista テストを行った．以降，このようなテストを Ballista テストにおける「完全テスト」と呼ぶこととする。完全テストの実行によって各関数のエラー分布を得ることができた．その中で，関数 sched_setscheduler のエラー分布を**図 1** に示す．

図中の Arg1，Arg2，Arg3 の各軸はそれぞれ第 1，第 2，第 3 引数のテスト値を表す．各テスト値を区別するために便宜上 1 から始まる通し番号を割り振っているが，それぞれの番号に対応するテスト値が存在する．図中の○はエラーとなった入力を示している．Ballista テストにおいてエラーとは OS やコンピュー

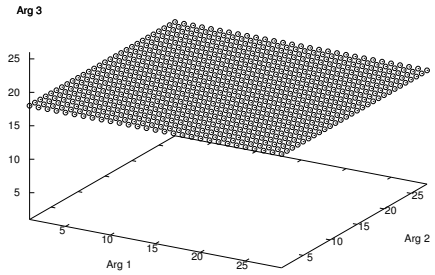

図 1 sched_setscheduler のエラー分布

タ自体の停止、タスクの停止、タスクの異常終了などを指す。第 3 引数が 18(解放した領域へのポインタ) の時は第 1, 第 2 引数の値に関わらずエラーになっている．このように 1 引数のみが寄与するエラーを 1 次エラーと呼ぶこととする。また，第 1 引数の値に関わらずエラーとなるような，2 引数が寄与するエラーを 2 次エラーと呼ぶこととする。

その他の関数についても同様にしてエラー分布を調べた．多くの関数で 1 次または 2 次のような低次エラーが存在することが確認された．文献 3) では 15 種類の POSIX OS に対して Ballista テストで発見されたエラーの分析を行っている．その結果，各 OS で発見された全エラーの内，平均で 81.75%ものエラーが 1 次エラーであることが報告されている．従って，少なくとも POSIX 関数に関しては 1 次エラーが多いと考えられる．また，Ballista テストで使用するテスト値の例外的・異常な値は境界値分析によって 1 次エラーの観点から選ばれていることも 1 次エラーが多い原因のひとつであろう。

Ballista テストの実行時間から 1 回のテスト当たりの平均実行時間を見積もることができる．今回のテストで 2,011,758 テストケースの実行に約 39 時間を要した．これは 1 秒当たり 14.3 テストケース，1 テストケース当たり 69.8ms となる計算である．(ただし，この中にはテスト結果をファイルに書き込む入出力の時間も含まれている) これにより，1 テストの実行時間は数 10〜数 100ms 程度と考えてよいであろう．

4. シャッフル法の提案

4.1 基本的なアイデア

各引数について一度実行した値は次回以降のテストケースに含めないというのが基本となる考え方である．シャッフル法はテスト対象となるモジュールが 1 次エラーを持つと仮定したときに効果的である．この効果を**図 2** を例に説明する．1 次エラーを持つ 2 引数モジュールの各引数に対して各 4 つのテスト値が準備されているとする (図 2 左)．1 回目のテストで図 2 右のように● (2,B) のテストケースが実行された時，次のテストケースとして 2 の列と B の列 (図の網掛け部分) をテストケース生成領域とすると，エラーを見つける確率は 1/6 である．他方，残りの部分からエラーを見つける確率は 3/9=1/3 である．

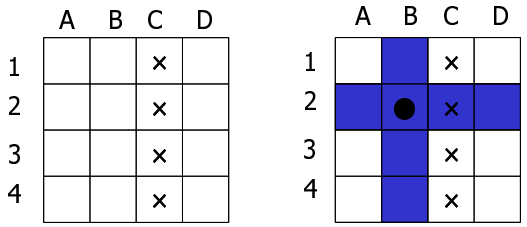

図 2 シャッフル法の基本的なアイデア

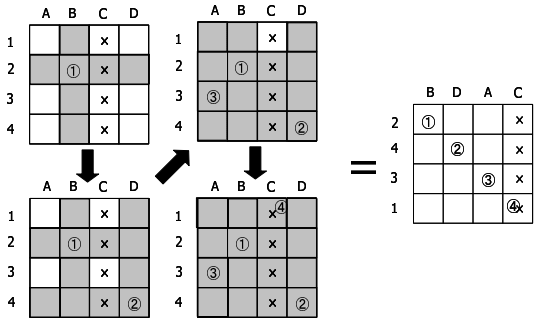

図 3 シャッフルの効果

一方でこれは，**図 3** のようにテストケースを生成するとき，各引数のテスト値をランダムに入れ替え (シャッフル) して，対角線上に位置するテストケースを順に実行することと等価である．よって，我々はこの手法をシャッフル法と名付けた．

4.2 シャッフル法のアルゴリズム

n 引数関数のモジュール $f(x_1, x_2, \cdots, x_n)$ において，$k(1 \leq k \leq n)$ 番目の引数 x_k は m_k 個の整数のテスト値を持ち，$1 \leq x_k \leq m_k$ とする．

(1) 各 k について，第 k 引数のテスト値を大きさ m_k の配列 A_k に格納する．テスト回数カウンタ count に 0 を代入する．$max_{k=1}^{n} m_k = M$ とする．

(2) count が M で割り切れれば (5) へ．

(3) $t_k = A_k[\text{count mod } m_k]$ とする．ただし (a mod b) は a を b で割った余りを表す．$T = (t_1, t_2, \cdots, t_n)$ が以前に実行されていなければ，

T を次に実行するテストケースとする．T が以前に実行されていれば，実行されていないテストケース候補が生成されるまでランダムにテストケースを生成し，最後に生成したテストケース候補 C を次に実行するテストケースとする．

(4) T または C を実行し，エラーが起きたならば，エラー発見を通知して終了する．そうでないならば T または C を実行済みであると記録し，count に 1 加える．(2) へ．

(5) 各 k に対し，$m_k \times l$ 回，以下の操作を繰り返す (シャッフル変数 l は 1 以上の整数値)．1 から m_k までの値の中からランダムに 2 個選び，i, j とする．A_k の i 番目の要素と j 番目の要素を交換する．
(3) へ．

尚、シャッフル変数 l は経験的に見て，3 であればよいであろう．5.2 において,「最初にエラーを発見するまでのテスト回数」を計測するが，l が 3 以下の場合，シャッフル回数が多くなるほどテスト回数を減少できたが，3 より大きくしてもテスト回数にはほとんど影響を与えないことが観察されている．

5. Ballista テストの高速化

5.1 ランダムテストの高速化

オリジナルの Ballista ツールにはテストケース生成順序に改善の余地がある．オリジナルの Ballista ツールでは，全テストケースの辞書的配列順から適当な間隔で間引くことによってテストケースを生成している．この方法では，テストの実行順序の方向が決まっているため，辞書的配列順の最後の方にエラーを起こすテストケースが固まって存在しているときにはエラーを速く見つけることはできない．

そこで，テストケースの生成順序を辞書的配列順からランダムな順序に変更し，ART のような既存のランダムテストに対する高速化手法を適用すれば，全てのテスト対象モジュールで見ると平均的には高速化できると考えられる．

Ballista テストでランダムテストを行うことを前提に，ランダムテストを高速化するために ART by Localization とシャッフル法の二つの手法を実装した．本節では Ballista の完全テストの結果に対して実験を行い，効果を比較・検証する．

5.2 ランダムテストとの比較

本節では，ART by Localization，シャッフル法をランダムテストと比較する．比較は「最初にエラーを発見するまでのテスト回数」と「テストケース生成のオーバーヘッド」の 2 点について行う．

1 回のテストの実行時間はテストケース生成時間と対象モジュール実行時間（実行の後処理やテスト結果の記録も含まれる）の二つの部分に分けることができる．テストケースの生成時間が対象モジュールの実行時間に比べ無視できる程度短い場合，最初のエラーを発見するまでのテスト回数を少なくできれば，テストを高速化できる．

最初にエラーを発見するまでのテスト回数

次のような実験を行い，ランダムテストと ART by Localization で最初にエラーを発見するまでのテスト回数の比較を行った．ただし，実際に Ballsita テストを実行するのではなく，3 節の結果を利用した．

テスト対象モジュールは 3 節で完全テストを行った 144 の POSIX 関数である．ランダムテストと ART by Localization のそれぞれでテストケースを生成し，3 節で得られた POSIX 関数に対する完全テストの結果と照合して，生成したテストケースがエラーを引き起こすかどうかをチェックする．このようにして最初にエラーを発見するまでテストケース生成を繰り返し，生成回数を数える．これを 1 試行とし，5000 試行繰り返して最初にエラーを発見するまでのテスト回数の平均値を求める．

ここで，ランダムテストと比較したテスト回数の削減率 (%) を定義する．ランダムテストでの最初にエラーを発見するまでの平均回数を $\bar{N}_{RT}$，ART by Localization またはシャッフル法では $\bar{N}_{ART}$ とすると，削減率 R は次の式で定義できる．

$$R = \frac{\bar{N}_{RT} - \bar{N}_{ART}}{\bar{N}_{RT}} \times 100$$

3 節で完全テストを行った関数の内，テストケースに占めるエラー入力の割合が 10 %以上の関数およびテストケースの総数が 100 以下の関数を除いた 42 個の関数について，削減率を D-ART by Localization，RRT by Localization，シャッフル法のそれぞれに対して求め，削減率の分布を示したのが**図 4** である．

D-ART by Localization の場合，半数近くの 19 個の関数で削減率がマイナスとなった．削減率が 10%を超えたのは 5 個ですべて 1 引数関数であった．これらのことより，D-ART by Localization ではほとんど高速化できないと言えるであろう．RRT by Localization の場合，4 個の関数で削減率がマイナスとなった．12 個の 1 引数関数は全て削減率が 15%を超えた．1 引数関数以外で 10%を超えたのは execvp と wctomb の 2 個のみであった．D-ART や RRT は本来、連続値を持つテスト値が定義域空間上に存在していること

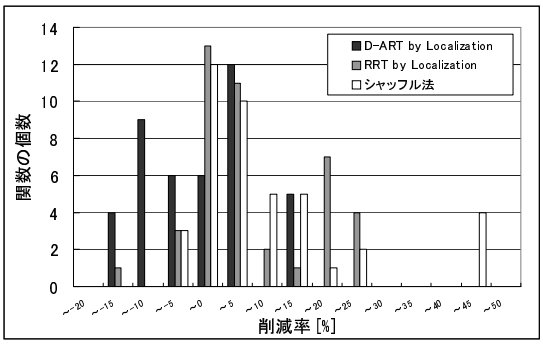

図 4 ART by Localization とシャッフル法の削減率の分布

を前提としている．Ballista のテスト値リストは離散的な値の集合であるので，テストケース間の距離には実質的な意味がないため，予想していたほどの効果がなかったと考えられる．

シャッフル法の場合，1 引数関数を除く全ての関数で最初にエラーを発見するまでのテスト回数を削減できた．1 引数関数の削減率は± 4%の範囲に収まった．これは有意な差であるとは言えず，1 引数関数に対してはランダムテストもシャッフル法も変わらないことを示している．sched_setscheduler, sched_getparam, sched_setparam，chown の 4 個の関数においては 50%近い削減率であった．これらの関数のエラー分布パターンは図 1 のような 1 次エラーの形をとっていた．このような場合には予測通り，シャッフル法が非常に有効であった．一方で，execvp のように 1 次エラーが複数，帯状に存在する場合には削減率はかなり下っている．

シャッフル法を ART by Localization と比較してみると，1 引数関数については RRT by Localization の方が高い削減率を示しているが，2 引数以上の関数については RRT by Localization と同等かそれ以上の効果を示していた．

テストケース生成のオーバーヘッド

エラーが全く存在しない，1,000,000 点から成る定義域空間において，10,000 個のテストケースを生成する時間を計測した (**図 5**, **図 6**)．この時間をオーバーヘッドの大きさを表す指標と考える．図の凡例中の n は引数の個数を表す．

図 5 によると D-ART，RRT いずれの場合もテストケース生成個数が多くなるに従って，1 テストケース当たりの生成時間も大きくなっている．2 引数関数で 20,000 個のテストケースを生成する時，D-ART では 168 秒，RRT では 207 秒であった．これは D-ART では 1 個当たりの生成時間では，8.4ms，RRT では

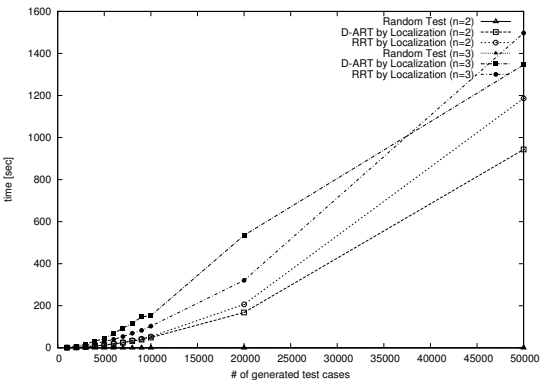

図 5 テストケース生成個数と生成時間の関係 (ART by Localization)

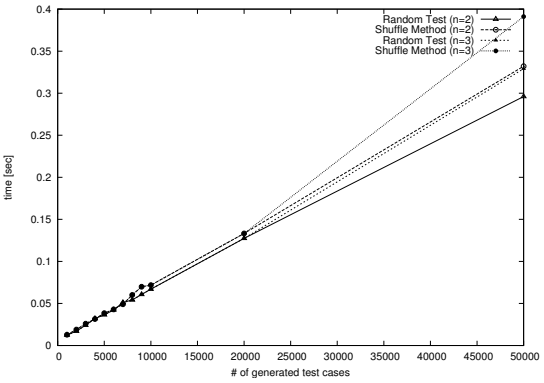

図 6 テストケース生成個数と生成時間の関係 (シャッフル法, ランダムテスト)

10.4ms となる計算である．対象モジュールの実行時間が 100ms であると仮定すると，テスト回数を 10%削減できても，テストケース生成のオーバーヘッドを加えるとほとんど高速化できなくなることになる．さらに，テストケースの生成個数が多くなれば，高速化とは逆効果になってしまう．つまり，全入力に占めるエラー入力の割合がある程度以下となるとテストケース生成のオーバーヘッドが無視できなくなり，ART by Localization では高速化どころか，むしろ遅くなってしまうことを意味する．ランダムテストは同じ条件の時，テストケース生成時間は 0.13 秒だったので 1 テストケース当たりの生成時間は 6.4 μs にしかならず，オーバーヘッドは ART の 1/1000 以下である．シャッフル法はランダムテストとほぼ同程度のオーバーヘッドであった．

ART by Localization では距離計算のオーバーヘッドよりも，領域分割などによるオーバーヘッドのほうが遥かに大きい．n 引数関数においては分割された部

分領域の個数はそれまでに実行したテスト回数の 2^n 倍である．例えば 3 引数関数において，1,000 回テストを実行したとすると，定義域空間は 8,000 個の部分領域に分割される．これらの部分領域はリストによって管理されているが，1 回テストする毎にこのリストに対して挿入処理を行わねばならない．ART by Localization では従来の ART と比べて，距離計算は大幅に簡略化さるが，分割された部分領域を管理するためのリストに対する挿入処理がオーバーヘッドの大部分を占める．一方，シャッフル法の場合，オーバーヘッドはシャッフル操作だけである．

6. おわりに

本研究では，Ballista テストを頑健性ランダムテストとして利用する際に，テストの高速化を図る目的で，ART by Localization とシャッフル法を実装し，その効果を比較した．

ART by Localization では主に 1 引数関数でしかテストケースを削減できなかったが，シャッフル法では 2 引数以上の関数で最大 50%のテストケース削減効果があることを示した．また、シャッフル法は ART by Localization と比較してテストケース生成のオーバーヘッドが非常に小さく，ランダムテストと比較しても大差がないことを示した．これらのことを総合的に考えると，POSIX 関数のような低次エラーが頻出するテスト対象モジュールに対しては ART by Localization よりもシャッフル法の方が優れていると言える．

今後の課題として，本研究で言及しなかった Mirror ART のような他の ART の実装，様々なエラー分布に対する最適な ART 手法の組合せを見つけることを挙げることができる．

参 考 文 献

1) Ivars Peterson, “ Fatal Defect: Chasing Killer Computer Bugs, ”Times Books, 1995.
2) P.Koopman and J.DeVale, “ The Exception Handling Effectiveness of POSIX Operating Systems, ”IEEE Transactions on Software Engineering, vol.26, no.9, pp.837-848, Sep. 2000.
3) J.Pan, “ The Dimensionality of Failures - A Fault Model for Characterizing Software Robustness, ”Proc. FTCS ’99, June 1999.
4) T.Y.Chen and D.H.Huang, “ Adaptive Random Testing by Localization, ”Proc. 11th Asia-Pacific Software Engineering Conference (APSEC’04), pp.292-298, Nov. 2004.