Unit Test Virtualization with VmVm

MD 輪講

博士後期課程 2 年　楊　嘉晨

大阪大学大学院コンピュータサイエンス専攻楠本研究室

2014 年 7 月 31 日（木）

著者情報と出典

Authors and Publication

Unit Test Virtualization with VMVM

訳 VMVM でユニットテストの仮想化

出典 ICSE 2014, **ACM Distinguished Paper**

著者 **Jonathan Bell**(PhD 学生), Gail Kaiser

所属 Computer Science, Columbia Univ.

Jonathan Bell 氏過去の研究[1][2][3]

---

[1] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. ``Chronicler: Lightweight Recording to Reproduce Field Failures''. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 362–371.

[2] Jonathan Bell, Swapneel Sheth, and Gail Kaiser. ``A Large-scale, Longitudinal Study of User Profiles in World of Warcraft''. In: *Proceedings of the 22nd International Conference on World Wide Web Companion*. 2013, pp. 1175–1184.

[3] Jonathan Bell, Swapneel Sheth, and Gail Kaiser. ``Secret Ninja Testing with HALO Software Engineering''. In: *Proceedings of the 4th International Workshop on Social Software Engineering*. ACM. 2011, pp. 43–47.

研究背景と目的

Background and Goal of the Research

1. テストスイートの総実行時間は**長――い** (数週間[4])
   → 従来最小化 (TSM) や優先付け (TSP) がある
   - TSM は NP 完全問題ですから近似法を利用
   - TSP は総実行時間が変わらない
   - テストの削減より障害を見逃す恐れがある

   → 視点を変える：そもそもテストの何処が遅い？
2. **大規模**なプロジェクトにおいてテストケース
   を**隔離して実行する**傾向がある (後述)
   (isolated)

目的 **隔離したテストケース**の総実行時間を**短縮** (isolated)

[4] Gregg Rothermel, Roland H Untch, et al. ``Test case prioritization: An empirical study''. In: *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.

研究背景と目的

Background and Goal of the Research

1. テストスイートの総実行時間は**長――い** (数週間[4])
   → 従来最小化 (TSM) や優先付け (TSP) がある

Test Suite Minimization (TSM)
テストスイート最小化

重複するテストケースを削減
カバレジに基づく研究が多い

2. **大規模**なプロジェクトにおいてテストケースを**隔離して実行する**(isolated)傾向がある (後述)

目的 **隔離したテストケース**(isolated)の総実行時間を**短縮**

[4] Gregg Rothermel, Roland H Untch, et al. ``Test case prioritization: An empirical study''. In: *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.

研究背景と目的

Background and Goal of the Research

1. テストスイートの総実行時間は**長――い** (数週間[4])
   → 従来最小化 (TSM) や優先付け (TSP) がある

Test Suite Prioritization (TSP)
テストスイート優先順位つけ

障害を起こしやすいテストケースを優先
修正された箇所に基づく研究が多い

2. **大規模**なプロジェクトにおいてテストケースを**隔離して実行する**(isolated)傾向がある (後述)

目的 **隔離した**(isolated)テストケースの総実行時間を**短縮**

[4] Gregg Rothermel, Roland H Untch, et al. ``Test case prioritization: An empirical study''. In: *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.

研究背景と目的

Background and Goal of the Research

1. テストスイートの総実行時間は**長――い** (数週間[4])
   → 従来最小化 (TSM) や優先付け (TSP) がある
   - TSM は NP 完全問題ですから近似法を利用
   - TSP は総実行時間が変わらない
   - テストの削減より障害を見逃す恐れがある

   → 視点を変える：そもそもテストの何処が遅い？

2. **大規模**なプロジェクトにおいてテストケースを**隔離して実行する**傾向がある (後述)

目的 **隔離したテストケースの総実行時間を短縮**

[4] Gregg Rothermel, Roland H Untch, et al. ``Test case prioritization: An empirical study''. In: *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.

研究背景と目的

Background and Goal of the Research

1. テストスイートの総実行時間は**長――い** (数週間[4])
   → 従来最小化 (TSM) や優先付け (TSP) がある
   - TSM は NP 完全問題ですから近似法を利用
   - TSP は総実行時間が変わらない
   - テストの削減より障害を見逃す恐れがある

   → 視点を変える：そもそもテストの何処が遅い？
2. **大規模**なプロジェクトにおいてテストケースを**isolated 隔離して実行する**傾向がある (後述)

目的 **isolated 隔離したテストケースの総実行時間を短縮**

[4] Gregg Rothermel, Roland H Untch, et al. ``Test case prioritization: An empirical study''. In: *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.

研究背景と目的

Background and Goal of the Research

1. テストスイートの総実行時間は**長――い** (数週間[4])
   → 従来最小化 (TSM) や優先付け (TSP) がある
   - TSM は NP 完全問題ですから近似法を利用
   - TSP は総実行時間が変わらない
   - テストの削減より障害を見逃す恐れがある

   → 視点を変える：そもそもテストの何処が遅い？
2. **大規模**なプロジェクトにおいてテストケースを**隔離して実行する**傾向がある (後述)

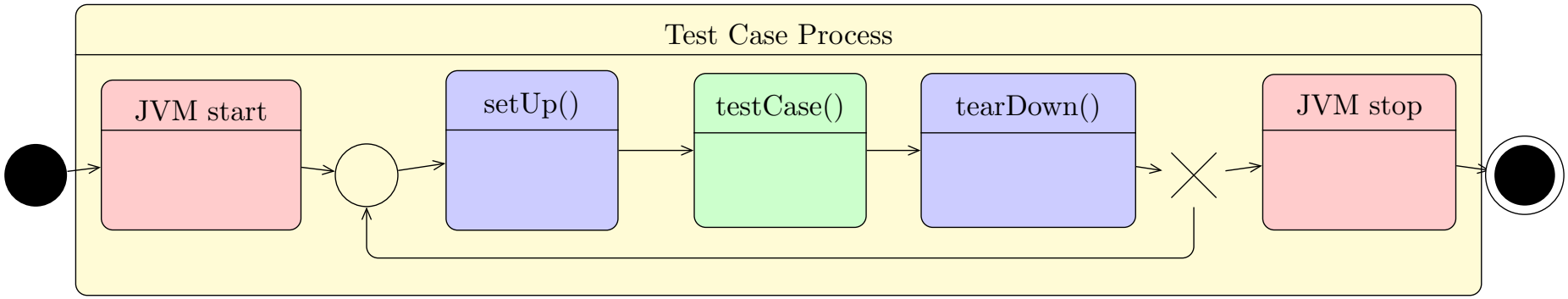
目的 **隔離したテストケースの総実行時間を短縮**

[4] Gregg Rothermel, Roland H Untch, et al. ``Test case prioritization: An empirical study''. In: *Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 179–188.

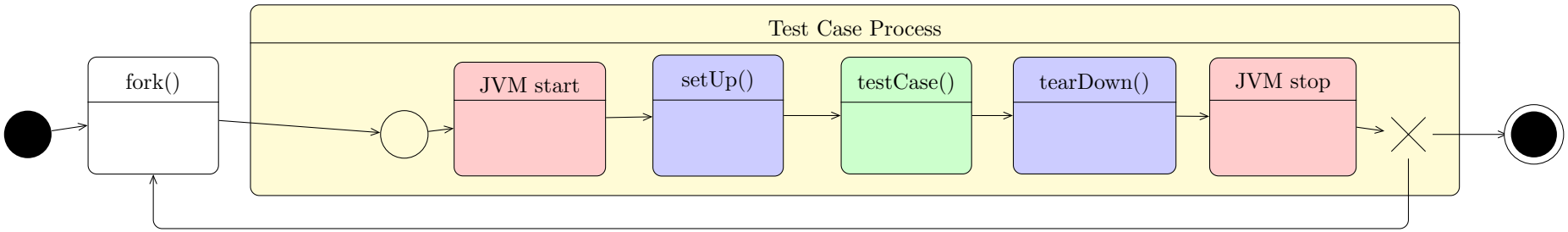
JUnit でテストケースの隔離実行

Isolated Test Suite in JUnit

(理想) 同じプロセス内で実行するテストスイート



(実際によくある) 隔離されたテストスイート

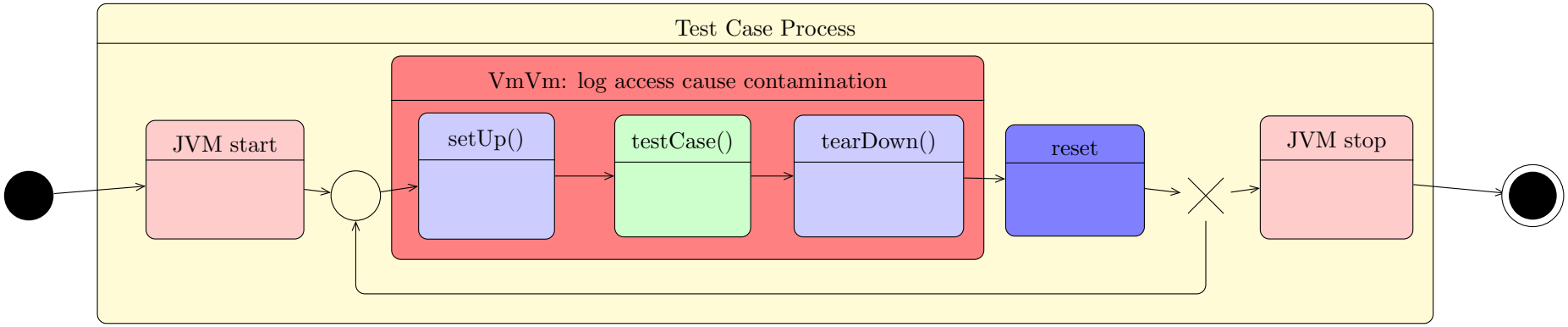


Ant/Maven 等の XML にオプションで切り替えられる

この研究の発想と貢献

Idea and Contribution of This Research

ユニットテストの仮想化



1. 1,200 OSS Java プロジェクトを調べて、大規模のプロジェクトにテストを隔離して実行する傾向があることを判明した (動機の調査)
2. ユニットテスト仮想化の手法を提案
3. Java で VmVm を実装して、障害を見逃す脅威を避けた上、実行時間の短縮を評価
   - 最大 97%(平均 62%) 性能の向上

動機の調査問題

Motivation Questions

MQ1 開発者はテストケースの実行を**隔離させるか**？

MQ2 **何故**開発者はテスト実行を隔離させるか？

MQ3 隔離テストの**オーバーヘッド**はどのぐらい？

Ohloh で「年間活躍開発者数」上位 1,200 の OSS Java プロジェクト (内 Ant/Maven+JUnit のは 591)

	Min	Max	Avg	Std dev
LOC	268	20,280.14k	519.40k	1,515.48k
Active Devs	3.00	350.00	15.88	28.49
Age (Years)	0.17	16.76	5.33	3.24

MQ1: 開発者はテスト隔離実行するか

MQ1: Do Developers Isolate Their Tests?

# of Tests in Project	# of Projects Creating New Processes Per Test	
0-10	24/71	(34%)
10-100	81/235	(34%)
100-1000	97/238	(41%)
>1000	38/47	(81%)
Lines of Code in Project	**# of Projects Creating New Processes Per Test**	
0-10k	7/42	(17%)
10k-100k	60/200	(30%)
100k-1m	115/267	(43%)
>1m	58/82	(71%)
All Projects	240/591	(41%)

MQ2: 隔離実行する原因は？

MQ2: Why Isolate Tests?

手書き tearDown は正確性を保証できない

文献[5]では Apache Commons CLI にほぼ 4 年が存続していたバグが隔離テストすれば浮上

tearDown で状態復元が難しい場合がある

```
1 public static final boolean ALLOW_EQUALS_IN_VALUE =
2   Boolean.valueOf(System.getProperty("org.apache.tomcat.util.http.
        ServerCookie.ALLOW_EQUALS_IN_VALUE", "false"))
3   .booleanValue();
4 /** Base Test case for {@link Cookies}. <b>Note</b> because of
5  * the use of <code>static final</code> constants in
6  * {@link Cookies}, each of these tests must be executed in a new
7  * JVM instance. The tests have been place in separate classes
8  * to facilitate this when running the unit tests via Ant. */
```

[5] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. ``Finding bugs by isolating unit tests''. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 496–499.

MQ3: 隔離実行のオーバーヘッドは？

MQ3: The Overhead of Isolation

MQ1 に収集したプロジェクトの内、50 近くは変更せずに直接ビルド・実行できる

中から規模と性質を考慮し 20 個を選びました

太字は元々隔離を指定したプロジェクト

Project	LOC (in k)	Test Classes	Overhead
Apache Ivy	305.99	119	342%
Apache Nutch	100.91	27	18%
Apache River	365.72	22	102%
Apache Tomcat	5692.45	292	42%
betterFORM	1114.14	127	377%
Bristlecone	16.52	4	3%
btrace	14.15	3	123%
Closure Compiler	467.57	223	888%
Commons Codec	17.99	46	407%
Commons IO	29.16	84	89%
Commons Validator	17.46	21	914%
FreeRapid Downloader	257.70	7	631%
gedcom4j	18.22	57	464%
JAXX	91.13	6	832%
Jetty	621.53	6	50%
JTor	15.07	7	1,133%
mkgmap	58.54	43	231%
Openfire	250.79	12	762%
Trove for Java	45.31	12	801%
upm	5.62	10	4,153%
Average	475.30k	56.4	618%

MQ3: 隔離実行のオーバーヘッドは？

MQ3: The Overhead of Isolation

MQ1 に収集したプロジェクトの内、50 近くは変更せずに直接ビルド・実行できる

中から規模と性質を考慮し 20 個を選びました

太字は元々隔離を指定したプロジェクト

Project	LOC (in k)	Test Classes	Overhead
Apache Ivy	305.99	119	342%
Apache Nutch	100.91	27	18%
Apache River	365.72	22	102%
Apache Tomcat	5692.45	292	42%
betterFORM	1114.14	127	377%
Bristlecone	16.52	4	3%
btrace	14.15	3	123%
Closure Compiler	467.47	222	8%
Commons Codec	17.99	46	407%
Commons IO	29.16	84	89%
Commons Validator	17.46	21	914%
FreeRapid Downloader	257.70	7	631%
gedcom4j	18.22	57	464%
JAXX	91.13	6	832%
Jetty	621.53	6	50%
JTor	15.07	7	1,133%
mkgmap	75.54	4	231%
Openfire	230.79	12	762%
Trove for Java	45.31	12	801%
upm	5.62	10	4,153%
Average	475.30k	56.4	618%

Bristlecone: 平均 20 秒

upm: 平均 0.15 秒

MQ3: 隔離実行のオーバーヘッドは？

MQ3: The Overhead of Isolation

MQ1 に収集したプロジェクトの内、50 近くは変更せずに直接ビルド・実行できる

中から規模と性質を考慮し 20 個を選びました

太字は元々隔離を指定したプロジェクト

Project	LOC (in k)	Test Classes	Overhead
Apache Ivy	305.99	119	342%
Apache Nutch	100.91	27	18%
Apache River	365.72	22	102%
Apache Tomcat	5692.45	292	42%
betterFORM	1114.14	127	377%
Bristlecone	16.52	4	3%
btrace	14.15	3	123%
Closure [illegible]	467.48	222	8%
Commons Codec	17.99	46	407%
Commons IO	29.16	84	89%
Commons Validator	17.46	31	914%
FreeRapid Downloader	257.70	7	631%
gedcom4j	15.52	5	464%
JAXX	91.13	6	832%
Jetty	621.55	7	5%
JTor	15.07	7	1,133%
mkgmap	75.54	41	231%
Openfire	230.79	12	762%
Trove for Java	45.31	12	801%
upm	5.62	10	4,153%
Average	475.30k	56.4	618%

Bristlecone: 平均 20 秒

テストケース毎に
実行時間が短いほど
オーバーヘッドが大きい

upm: 平均 0.15 秒

動機の調査問題への回答

Answers to Motivation Questions

MQ1 開発者はテストケースの実行を隔離させるか

→ 全体の 41%, 大規模の 81%(テスト数)71%(行数)

MQ2 何故開発者はテスト実行を隔離させるか

→ 手書き `tearDown` は正確性を保証できない、状態復元が難しい場合がある

MQ3 隔離テスト実行のオーバーヘッドは

→ 平均 618%, 最大 4,153%

結論: 大規模のプロジェクトにテストを隔離して実行する傾向がある、オーバーヘッドが大きい

提案手法の流れ

Approach

静的検査と bytecode の書き換え

入力： 

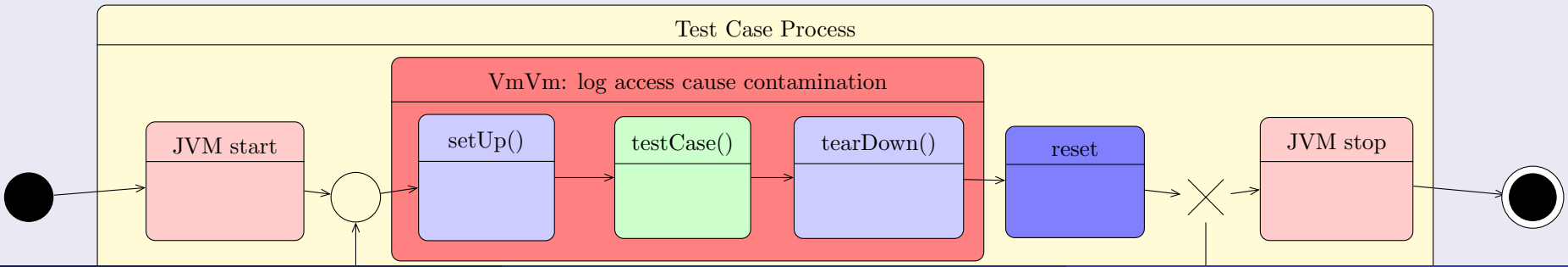
と 

静的検査 → メモリ領域を安全性で分類

instrument
bytecode を書き換え → 出力：書き換えた 

instrumented

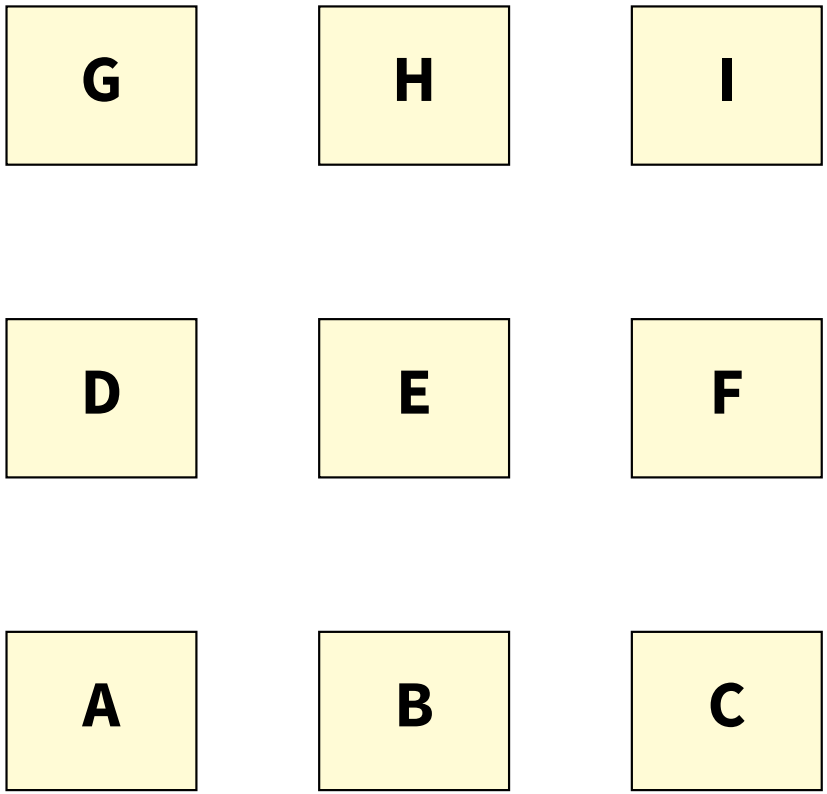
動的仮想マシン



提案手法のイメージ

Image of the Approach

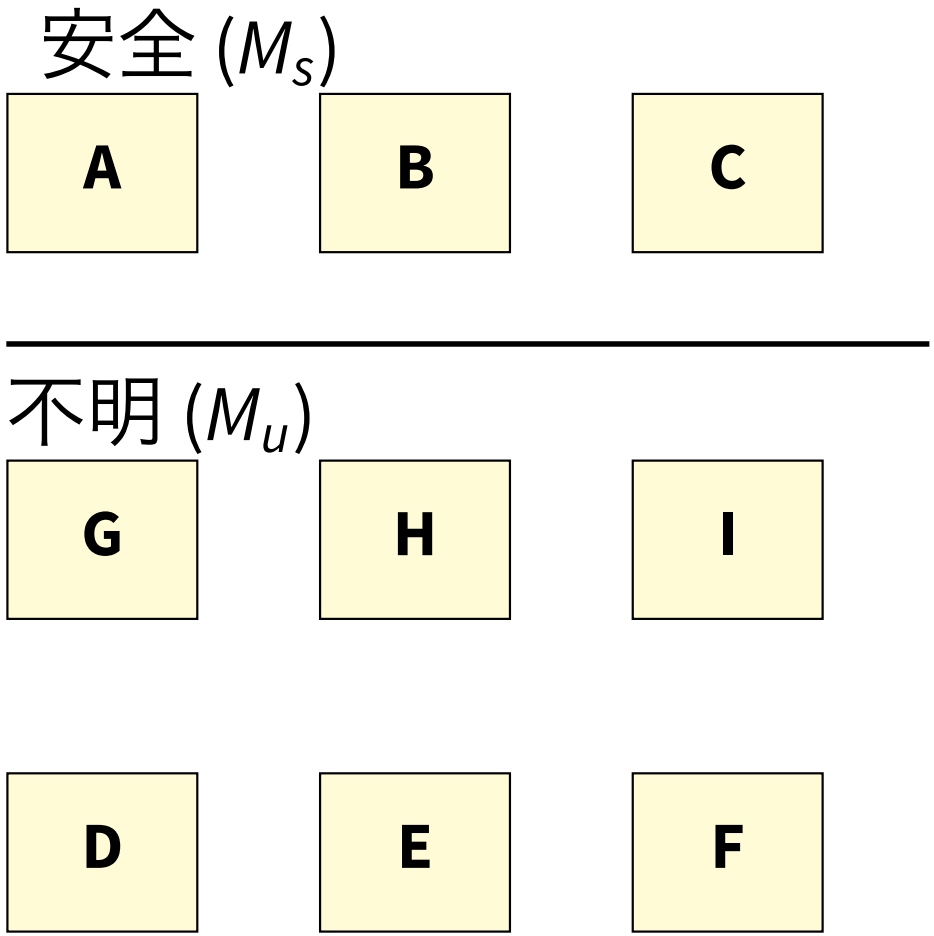
- 全メモリ領域
- 静的検査して安全 M_s と不明 M_u に分ける
- テスト 1 を実行して、Contaminated 汚染されたメモリ領域を記録
- テスト 2 を実行し、Contaminated 前に汚染されたメモリ領域をアクセス直前に re-initialization 再度初期化
- ...



提案手法のイメージ

Image of the Approach

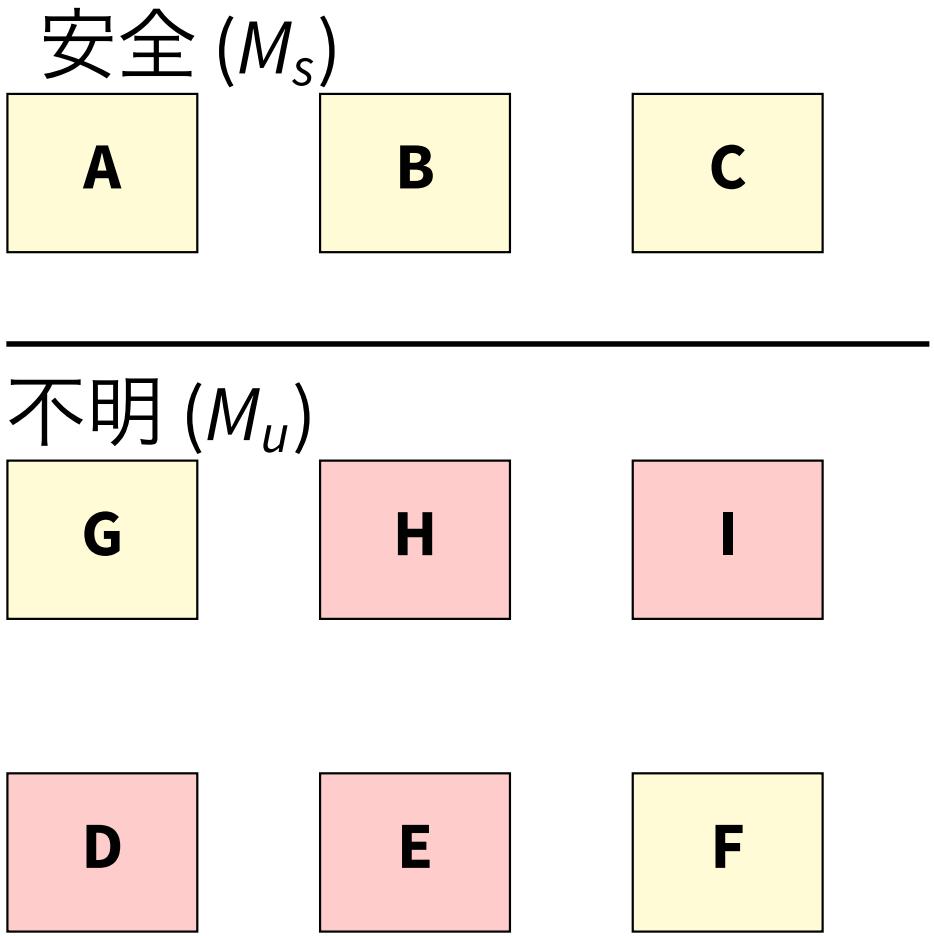
- 全メモリ領域
- 静的検査して安全 M_s と不明 M_u に分ける
- テスト１を実行して、Contaminated 汚染されたメモリ領域を記録
- テスト２を実行し、Contaminated 前に汚染されたメモリ領域をアクセス直前に re-initialization 再度初期化
- ...



提案手法のイメージ

Image of the Approach

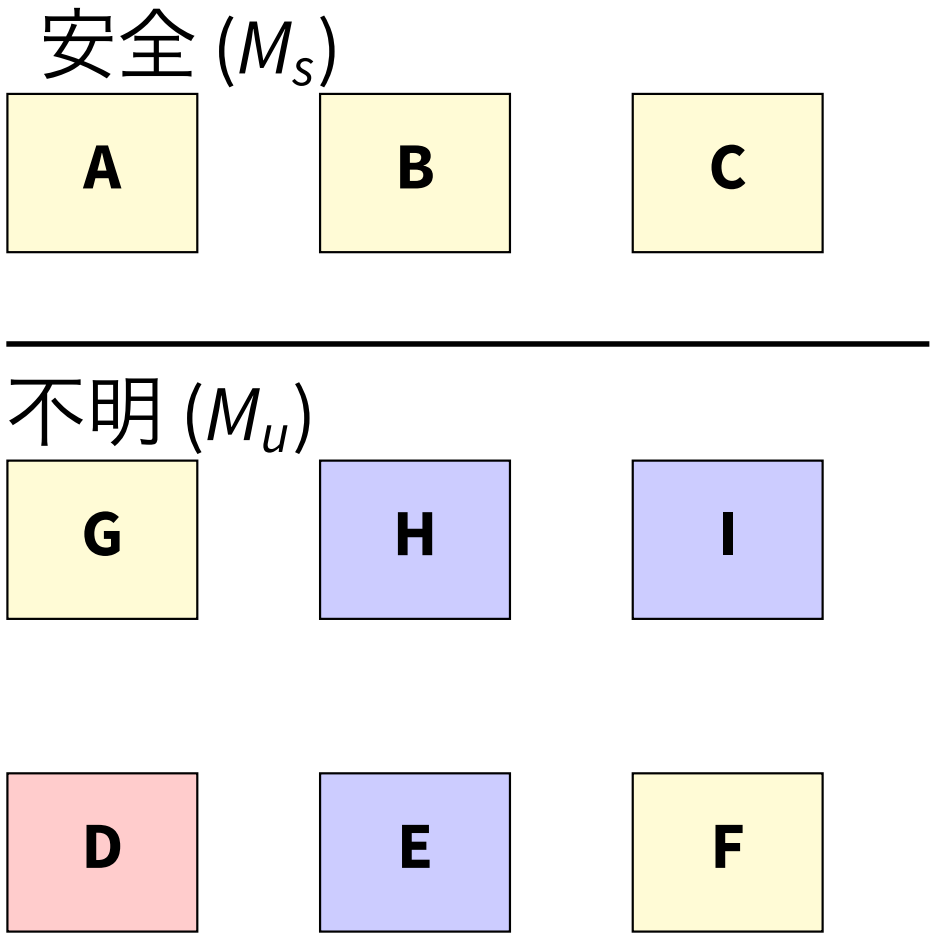
- 全メモリ領域
- 静的検査して安全 M_s と不明 M_u に分ける
- テスト 1 を実行して、Contaminated 汚染されたメモリ領域を記録
- テスト 2 を実行し、Contaminated 前に汚染されたメモリ領域をアクセス直 re-initialization 前に再度初期化
- ...



提案手法のイメージ

Image of the Approach

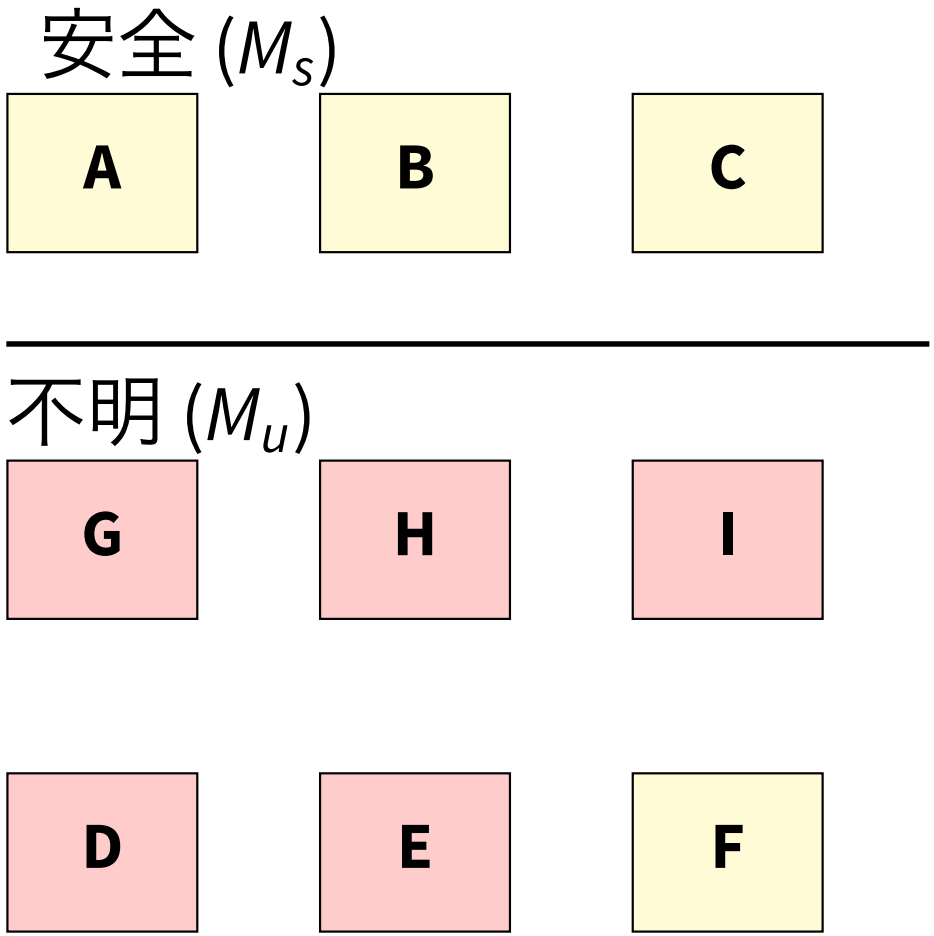
- 全メモリ領域
- 静的検査して安全 M_s と不明 M_u に分ける
- テスト 1 を実行して、Contaminated 汚染されたメモリ領域を記録
- テスト 2 を実行し、Contaminated 前に汚染されたメモリ領域をアクセス直前に re-initialization 再度初期化
- ...



提案手法のイメージ

Image of the Approach

- 全メモリ領域
- 静的検査して安全 M_s と不明 M_u に分ける
- テスト 1 を実行して、Contaminated 汚染されたメモリ領域を記録
- テスト 2 を実行し、Contaminated 前に汚染されたメモリ領域をアクセス直前に re-initialization 再度初期化
- ...

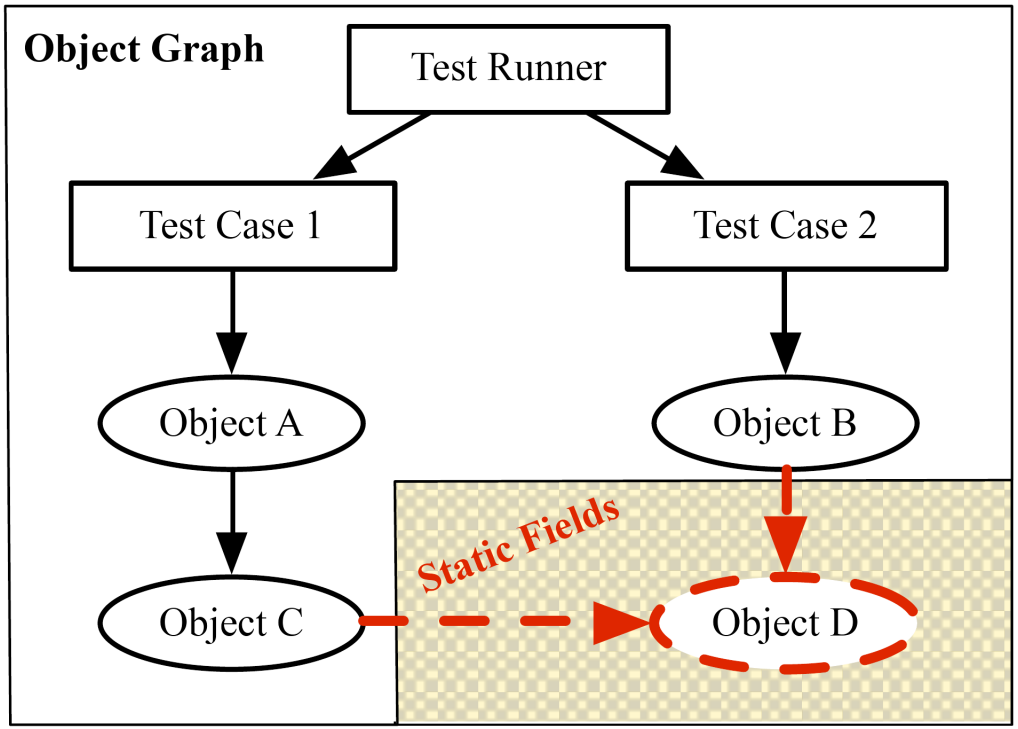


Java メモリ管理の背景

Java Memory Management Background

安全 (M_s) なメモリ:
スタックのメモリ
(ローカル変数、引数)

JUnit より保証:
テストケース間にオブジェクトを渡しない



チェックする必要: クラスにある static フィルドのみ

静的検査

Offline Analysis

```
1 public class SafeStaticExample{
2   public static final String s = "abcd";
3   public static final int x = 5;
4   public static final int y = x * 3;
5 }
```

相当する bytecode のイメージ:

```
1 public class SafeStaticExample{
2   public static void <clinit>(){
3     SafeStaticExample.s = "abcd";
4     SafeStaticExample.x = 5;
5     SafeStaticExample.y = x * 3;
6   }
7 }
```

安全な static 変数:

- `final`
  immutable type
- 不可変型
- 値が定数のみ依存する

安全性はクラス単位で判定

一個以上の M_u フィールドを持つとクラス全体は M_u になる

Bytecode の書き換え (1/2)

Bytecode Instrumentation (1/2)

プログラムと外部ライブラリ (JRE と JUnit を除く) にある全クラスに対して、M_u にあるクラスへの初期化操作の bytecode を書き換え (Instrument):

1. 新しいインスタントを作る
2. クラスの static メソットを呼び出し
3. クラスの static フィルドへアクセス
4. リフレクション (reflection) で直接初期化

Native コードからのアクセスに関しては、実験対象 (5 章) では必要ない

Bytecode の書き換え (2/2)

Bytecode Instrumentation (2/2)

書き換えたbytecode (instrument) で２つのことを行う：

- 初期化されたかをログに記録
  - クラス内に static フィルドを追加
  - VmVm の VirtualRuntime 内に
- 前のテストで汚れたクラスを再度初期化

著者らは JRE の API を手作業で全部検証した

- 48 個の不安全なクラスを見つけた
- VmVm のラッパー (wrapper) に置換し、copy-on-write 機能を実装

テスト自動化との統合

Test Automation Integration

ant との統合

```
1 <classpath>...</classpath>
2 <formatter classname="edu.
        columbia.cs.psl.vmvm.
        AntJUnitTestListener"
        extension=".xml"/>
3 <jvmarg value="-Xbootclasspath/
        a:vmvm.jar"/>
4
```

他のツール

```
1 VirtualRuntime.reset();
```

maven との統合

```
1 <additionalClasspathElements>
        ...
2 </additionalClasspathElements>
3 <properties><property>
4 <name>listener</name>
5 <value>
6 edu.columbia.cs.psl.vmvm.
        MvnVMVMListener
7 </value>
8 </property></properties>
9
```

評価実験の設定

Setups of Experimental Evaluations

比較対象: 1. TSM 手法 2. プロセスレベルテスト隔離
Reduction in Time(RT) Reduction in Fault-finding(RF)

評価指標: 1. 時間短縮 2. 欠陥発見損失

TSM 手法に関して既存研究[6]で評価された最も有効な手法[7]を用いて、既存の評価対象 SIR[8]で行う

プロセスレベルテスト隔離に関して、MQ3 で使われている 20 個のプロジェクトで評価

Ubuntu 12.04.1 LTS, Java 1.7.0 25, 4-core 2.66Ghz Xeon, 8GB RAM

[6] Lingming Zhang et al. ``An empirical study of junit test-suite reduction''. In: *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE. 2011, pp. 170–179.

[7] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. ``A methodology for controlling the size of a test suite''. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.3 (1993), pp. 270–285.

[8] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. ``Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact''. In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435.

実験 1: TSM と SIR での比較[9]

Application	LOC (in k)	Test Classes	TSM *RS*	TSM *RT*	VMVM *RT*	Combined *RT*
Ant v1	25.83k	34	3%	4%	39%	40%
Ant v2	39.72k	52	0%	0%	36%	37%
Ant v3	39.80k	52	0%	1%	36%	37%
Ant v4	61.85k	101	7%	4%	34%	37%
Ant v5	63.48k	104	6%	11%	25%	26%
Ant v6	63.55k	105	6%	11%	26%	27%
Ant v7	80.36k	150	11%	21%	28%	38%
Ant v8	80.42k	150	10%	18%	27%	37%
JMeter v1	35.54k	23	8%	2%	42%	42%
JMeter v2	35.17k	25	4%	1%	41%	42%
JMeter v3	39.29k	28	11%	5%	44%	48%
JMeter v4	40.38k	28	11%	5%	42%	47%
JMeter v5	43.12k	32	16%	8%	50%	52%
jtopas v1	1.90k	10	13%	34%	75%	77%
jtopas v2	2.03k	11	11%	31%	70%	76%
jtopas v3	5.36k	18	17%	27%	48%	68%
xml-sec v1	18.30k	15	33%	22%	69%	73%
xml-sec v2	18.96k	15	33%	26%	79%	80%
xml-sec v3	16.86k	13	38%	19%	54%	55%
Average	37.47k	51	12%	13%	46%	49%

Reduction in Time(RT)

時間短縮に関して圧勝

Reduction in Fault-finding(RF)

欠陥発見損失に関して、
実験で両方共 0% が、
既存研究 [9] より
TSM が 100% 場合がある
つまり本来あるべきの欠陥を
一つも見つけず
VMVM は常に 0%

VMVM と TSM を組み合わせて使うのも可能

[9] Gregg Rothermel, Mary Jean Harrold, et al. ``An empirical study of the effects of minimization on the fault detection capabilities of test suites''. In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, pp. 34–43.

プロセスレベルテスト隔離と比較

Study2: More Applications

Project	Revisions	LOC (in k)	Age (Years)	# of Tests		Overhead			False Positives	
				Classes	Methods	VMVM	Forking	RT	VMVM	No Isolation
Apache Ivy	1233	305.99	5.77	119	988	48%	342%	67%	0	52
Apache Nutch	1481	100.91	11.02	27	73	1%	18%	14%	0	0
Apache River	264	365.72	6.36	22	83	1%	102%	50%	0	0
Apache Tomcat	8537	5,692.45	12.36	292	1,734	2%	42%	28%	0	16
betterFORM	1940	1,114.14	3.68	127	680	40%	377%	71%	0	0
Bristlecone	149	16.52	5.94	4	39	6%	3%	-3%	0	0
btrace	326	14.15	5.52	3	16	3%	123%	54%	0	0
Closure Compiler	2296	467.57	3.85	223	7,949	174%	888%	72%	0	0
Commons Codec	1260	17.99	10.44	46	613	34%	407%	74%	0	0
Commons IO	961	29.16	6.19	84	1,022	1%	89%	47%	0	0
Commons Validator	269	17.46	6.19	21	202	81%	914%	82%	0	0
FreeRapid Downloader	1388	257.70	5.10	7	30	8%	631%	85%	0	0
gedcom4j	279	18.22	4.44	57	286	141%	464%	57%	0	0
JAXX	44	91.13	7.44	6	36	42%	832%	85%	0	0
Jetty	2349	621.53	15.11	6	24	3%	50%	31%	0	0
JTor	445	15.07	3.94	7	26	18%	1,133%	90%	0	0
mkgmap	1663	58.54	6.85	43	293	26%	231%	62%	0	0
Openfire	1726	250.79	6.44	12	33	14%	762%	87%	0	0
Trove for Java	193	45.31	11.86	12	179	27%	801%	86%	0	0
upm	323	5.62	7.94	10	34	16%	4,153%	97%	0	0
Average	1356.3	475.30	7.32	56.4	717	34%	618%	62%	0	3.4
Average (Isolated)	1739.3	743.16	8.86	58.7	419	12%	648%	56%	0	6.8
Average (Not Isolated)	973.3	207.43	5.79	54.1	1,015	57%	588%	68%	0	0

プロセスレベルテスト隔離と比較

Study2: More Applications

Project	Revisions	LOC (in k)	Age (Years)	# of Tests: Classes	# of Tests: Methods	Overhead: VMVM	Overhead: Forking	RT	False Positives: VMVM	False Positives: No Isolation
Apache Ivy	1233	305.99	5.77	119	988	48%	342%	67%	0	52
Apache Nutch	1481	100.91	11.02	27	73	1%	18%	14%	0	0
Apache River	264	365.72	6.36	22	83	1%	102%	50%	0	0
Apache Tomcat	8537	5,692.45	12.36	292	1,734	2%	42%	28%	0	16
betterFORM	1940	1,114.14	3.68	127	680	40%	377%	71%	0	0
Bristlecone	149	16.52	5.94	4	39	6%	3%	-3%	0	0
btrace	326	14.15	5.52	3	16	3%	123%	54%	0	0
Closure Compiler	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
Commons Codec	[illegible]	17.99	10.44	[illegible]	[illegible]	[illegible]	[illegible]	74%	0	0
[illegible]	[illegible]	[illegible]	[illegible]	84	[illegible]	[illegible]	89%	[illegible]	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	[illegible]
FreeRapid Downloader	1388	257.70	5.10	7	30	8%	631%	85%	0	0
gedcom4j	279	18.22	4.44	57	286	141%	464%	57%	0	0
JAXX	44	91.13	7.44	6	36	42%	832%	85%	0	0
Jetty	2349	621.53	15.11	6	24	3%	50%	31%	0	0
JTor	445	[illegible]	[illegible]	7	[illegible]	[illegible]	[illegible]	[illegible]	0	0
mkgmap	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
Openfire	1726	250.79	6.44	12	33	14%	762%	87%	0	0
Trove for Java	193	45.31	11.86	12	179	27%	801%	86%	0	0
upm	323	5.62	7.94	10	34	16%	4,153%	97%	0	0
Average	1356.3	475.30	7.32	56.4	717	34%	618%	62%	0	3.4
Average (Isolated)	1739.3	743.16	8.86	58.7	419	12%	648%	56%	0	6.8
Average (Not Isolated)	973.3	207.43	5.79	54.1	1,015	57%	588%	68%	0	0

Bristlecone: RT:-3%, Fork より遅くなる
実行時間が長いテストケースに対して嬉しくない

upm: RT: 97%, 36 倍早くなった

プロセスレベルテスト隔離と比較

Study2: More Applications

Project	Revisions	LOC (in k)	Age (Years)	# of Tests: Classes	# of Tests: Methods	Overhead: VMVM	Overhead: Forking	Overhead: RT	False Positives: VMVM	False Positives: No Isolation
Apache Ivy	1233	305.99	5.77	119	988	48%	342%	67%	0	52
Apache Nutch	1481	100.91	11.02	27	73	1%	18%	14%	0	0
Apache River	264	365.72	6.36	22	83	1%	102%	50%	0	0
Apache Tomcat	8537	5,692.45	12.36	292	1,734	2%	42%	28%	0	16
betterFORM	1940	1,114.14	3.68	127	680	40%	377%	71%	0	0
Bristlecone	149	16.52	5.94	4	39	6%	3%	-3%	0	0
btrace	326	14.15	5.52	3	16	3%	123%	54%	0	0
Closure Compiler	2296	467.57	3.85	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
Commons IO	961	29.16	6.19	84	1,022	1%	89%	47%	0	0
Commons Validator	269	17.46	6.19	21	202	81%	914%	82%	0	0
FreeRapid Downloader	1388	257.70	5.10	7	30	8%	631%	85%	0	0
gedcom4j	279	18.22	4.44	57	286	141%	464%	57%	0	0
JAXX	44	91.13	7.44	6	36	42%	832%	85%	0	0
Jetty	2349	621.53	15.11	6	24	3%	50%	31%	0	0
JTor	445	15.07	3.94	7	26	18%	1,133%	90%	0	0
mkgmap	1663	58.54	6.85	43	293	26%	231%	62%	0	0
Openfire	1726	250.79	6.44	12	33	14%	762%	87%	0	0
Trove for Java	193	45.31	11.86	12	179	27%	801%	86%	0	0
upm	323	5.62	7.94	10	34	16%	4,153%	97%	0	0
Average	1356.3	475.30	7.32	56.4	717	34%	618%	62%	0	3.4
Average (Isolated)	1739.3	743.16	8.86	58.7	419	12%	648%	56%	0	6.8
Average (Not Isolated)	973.3	207.43	5.79	54.1	1,015	57%	588%	68%	0	0

FP を起こらず、Fork と同じ隔離の効果

手法の制限と妥当性への脅威

Limitations and Threats to Validity

実験対象の選択は妥当であるか？

→ SIR はよく研究されている、SIR より大規模な対象も実験した

→ Ohloh で最も大きいプロジェクトを対象にした

利用できる場面はテスト間で準備の時間が長い場合

→ インタラクションに基づくテストに向いてないかも

→ TSM や TSP と組み合わせて利用できる

プログランキング言語に依存するか？

→ メモリ管理された言語 (Memory Managed Language) かつユニットテスト環境があれば十分

→ 評価結果は Java 依存かも (隔離テストが必要な場面が多い)

VmVm の仮想マシンとしての隔離が十分であるか？

→ メモリ状態とユニットテストのみ

→ ファイルやデータベースへのアクセスも隔離する手法があればいいが、主旨を超えている

結論と今後の課題

Conclusions and Future Work

結論

- 1,200 最も大きい Java プロジェクトを調べて、**40%(内大規模の 81%)** はテストケースを**隔離実行**していることを判明
- ユニットテスト仮想化手法を提案し、VMVM を実装し、隔離性を維持した上、**最大 97%(平均 62%)** のテスト時間を短縮

今後の課題

- 他の言語に対応

→ メモリ管理されない C や同じ JVM の Scala 等

- 実装に改善の余地がある

所感

- ☺ 大規模なプロジェクトにテストケースを隔離して実行する傾向を見破った視点が鋭い
- ☺ 実装がわりと簡単ですが、効果が抜群
- ☺ 評価はしっかり、スケールは半ばない
- ☺ さすが ICSE Distinguished Paper 感
- ☹ 書き方は大袈裟 (騙された！→っでもすごい！)
  - ☹ 軽量級仮想マシン → Bytecode 書き換え (instrument)
  - ☹ 1,200(集まった) → 591(XML 解析できた) → 50 近く (直接実行できた) → 20(評価に使った)
- ☹ SIR で TSM との比較ですが、既存手法は隔離実行を考慮してない
- ☹ 隔離実行は並列できるが、提案手法はできない