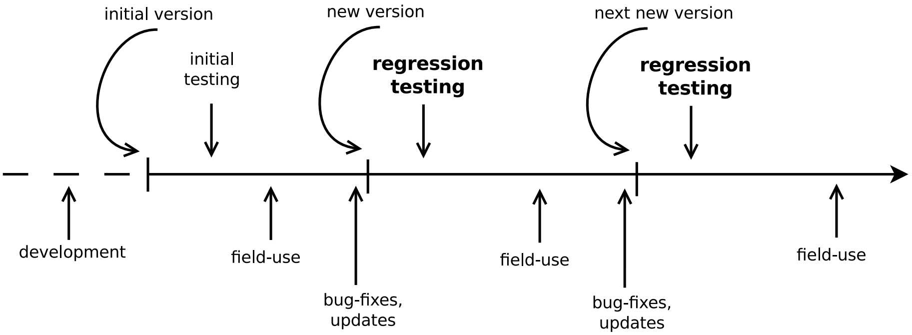
Encoding Test Requirements as Constraints for Test Suite Minimization

José Campos and Prof. Rui Abreu

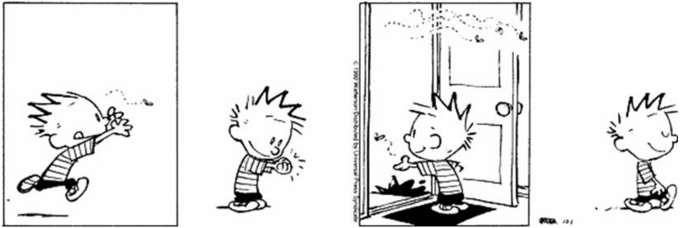
University of Porto, Portugal
https://www.fe.up.pt/

April 16th, 2013

The Life of a Software System

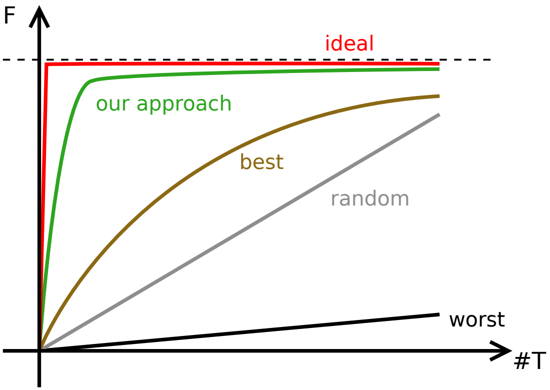


Motivation I - More Fixs More Bugs



Software (regression) testing is performed to guarantee that changes did not affect the system negatively.

Motivation II - More Faults More Earlier



Software (regression) testing is performed to detect errors as early as possible.

Related Work

- Chavatal [6] uses a simple greedy heuristic.
- Offutt, Pan and Voas [16] presented a heuristics to reduce test set sizes based on reordering the test execution sequence.
- Harrold, Gupta and Soffa [13] developed a heuristic based on a determined number of test case covering specific demand.
- Tallam and Gupta [18] developed the Delayed-Greedy approach.
- Jeffrey and Gupta [15] extended the HGS heuristic which that certain test cases are selectively retained.

Related Work

- Chavatal [6] uses a simple greedy heuristic.
- Offutt, Pan and Voas [16] presented a heuristics to reduce test set sizes based on reordering the test execution sequence.
- Harrold, Gupta and Soffa [13] developed a heuristic based on a determined number of test case covering specific demand.
- Tallam and Gupta [18] developed the Delayed-Greedy approach.
- Jeffrey and Gupta [15] extended the HGS heuristic which that certain test cases are selectively retained.
- Black, Melachrinoudis and Kaeli [4] considered a bi-criteria approach that takes into account minimizing a test suite and maximize error detection rates.

Related Work

- Chavatal [6] uses a simple greedy heuristic.
- Offutt, Pan and Voas [16] presented a heuristics to reduce test set sizes based on reordering the test execution sequence.
- Harrold, Gupta and Soffa [13] developed a heuristic based on a determined number of test case covering specific demand.
- Tallam and Gupta [18] developed the Delayed-Greedy approach.
- Jeffrey and Gupta [15] extended the HGS heuristic which that certain test cases are selectively retained.
- Black, Melachrinoudis and Kaeli [4] considered a bi-criteria approach that takes into account minimizing a test suite and maximize error detection rates.
- Hsu and Orso [14] approach is based on encoding the user-provide minimization problem and related criteria as binary Integer Linear Programming problem.

Concepts I

Definition (Program)

A program Π is a collection of M components, $M = \{m_1, \ldots, m_j, \ldots, m_M\}$, implementing a specific set of specifications and requirements.

Concepts I

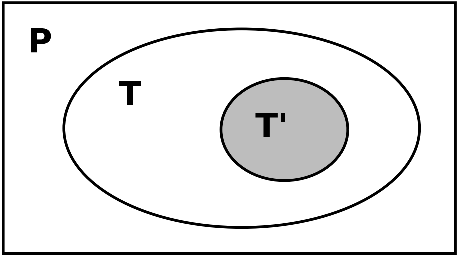
Definition (Program)

A program Π is a collection of M components, $M = \{m_1, \ \ldots, \ m_j, \ \ldots, \ m_M\}$, implementing a specific set of specifications and requirements.

Definition (Test Suite)

A test suite $T = \{t_1, \ldots, t_i, \ldots, t_N\}$ is a set of N test cases that are intended to test whether the program follows the specified set of requirements.

Problem



Find a representative set, T', of test cases from T that satisfies all m_js.

Example

M	Program: Calculator
	`public static class Calculator`
	`{`
m_1	`public int add(int x, int y) { return x + y;}`
m_2	`public int sub(int x, int y) { return x - y;}`
m_3	`public int mul(int x, int y) { return x * y;}`
	`}`

Example - Adding Tests I

M	Program: Calculator	T t_1
	`public static class Calculator`	
	`{`	
m_1	`public int add(int x, int y) { return x + y;}`	1
m_2	`public int sub(int x, int y) { return x - y;}`	1
m_3	`public int mul(int x, int y) { return x * y;}`	0
	`}`	

```
public class t1 {
 public int testAdd() { assertTrue(Calculator.add(1, 2) == 3); }
 public int testSub() { assertTrue(Calculator.sub(2, 1) == 1); }
}
```

Example - Adding Tests II

M	Program: Calculator	T t_1	t_2
	`public static class Calculator`		
	`{`		
m_1	`public int add(int x, int y) { return x + y;}`	1	1
m_2	`public int sub(int x, int y) { return x - y;}`	1	0
m_3	`public int mul(int x, int y) { return x * y;}`	0	0
	`}`		

```
public class t2 {
 public int testAdd() { assertTrue(Calculator.add(1, 0) == 1); }
}
```

Example - Adding Tests II

M	Program: Calculator	T t_1	t_2
	`public static class Calculator`		
	`{`		
m_1	`public int add(int x, int y) { return x + y;}`	1	1
m_2	`public int sub(int x, int y) { return x - y;}`	1	0
m_3	`public int mul(int x, int y) { return x * y;}`	0	0
	`}`		

```
public class t2 {
 public int testAdd() { assertTrue(Calculator.add(1, 0) == 1); }
}
```

• • •

Coverage Matrix

$$\begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Coverage Matrix

$$\begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Constraint satisfaction problem are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations.

Coverage Matrix as Constraints

$$\begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \qquad \rightarrow c_1 = (t_1 \vee t_2)$$

Coverage Matrix as Constraints

$$\begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array} \begin{array}{c} \begin{matrix} t_1 & t_2 & t_3 & t_4 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \qquad \begin{array}{l} \rightarrow c_1 = (t_1 \vee t_2) \\ \rightarrow c_2 = (t_1 \vee t_3) \end{array}$$

Coverage Matrix as Constraints

$$
\begin{array}{c c}
 & \begin{matrix} t_1 & t_2 & t_3 & t_4 \end{matrix} \\
\begin{matrix} m_1 \\ m_2 \\ m_3 \end{matrix} & \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{array}
\qquad
\begin{matrix} \rightarrow c_1 = (t_1 \vee t_2) \\ \rightarrow c_2 = (t_1 \vee t_3) \\ \rightarrow c_3 = (t_4) \end{matrix}
$$

Coverage Matrix as Constraints

$$
\begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \end{array}
\begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}
\qquad
\begin{array}{l} \\ \rightarrow c_1 = (t_1 \vee t_2) \\ \rightarrow c_2 = (t_1 \vee t_3) \\ \rightarrow c_3 = (t_4) \end{array}
$$

$$C = (\ (t_1 \vee t_2) \wedge (t_1 \vee t_3) \wedge (t_4)\ )$$

Solving Constraints

$$C = (\ (t_1 \vee t_2) \wedge (t_1 \vee t_3) \wedge (t_4)\ )$$

Solving Constraints

$$C = (\ (t_1 \lor t_2) \land (t_1 \lor t_3) \land (t_4)\ )$$

- $\{t_1, t_4\}$
- $\{t_1, t_2, t_4\}$
- $\{t_1, t_3, t_4\}$
- $\{t_2, t_3, t_4\}$
- $\{t_1, t_2, t_3, t_4\}$

Solving Constraints

$$C = (\ (t_1 \vee t_2) \wedge (t_1 \vee t_3) \wedge (t_4)\ )$$

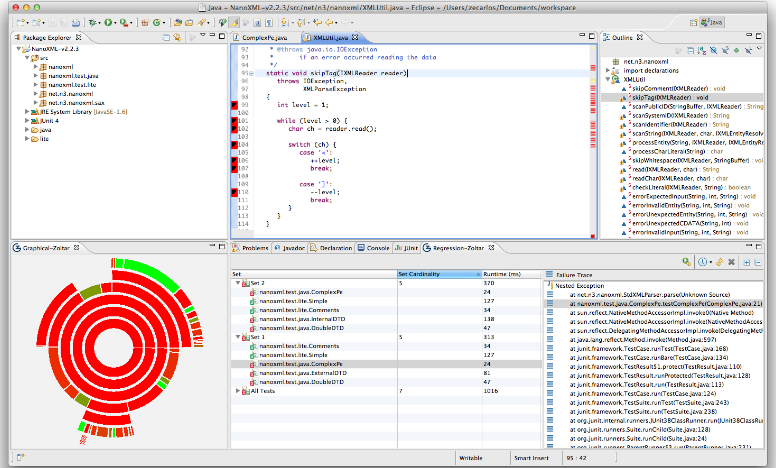
- $\{t_1, t_4\}$
- $\{t_1, t_2, t_4\}$
- $\{t_1, t_3, t_4\}$
- $\{t_2, t_3, t_4\}$
- ~~$\{t_1, t_2, t_3, t_4\}$~~

Solving Constraints

$$C = (\ (t_1 \vee t_2) \wedge (t_1 \vee t_3) \wedge (t_4)\ )$$

- $\{t_1, t_4\}$
- ~~$\{t_1, t_2, t_4\}$~~
- ~~$\{t_1, t_3, t_4\}$~~
- $\{t_2, t_3, t_4\}$
- ~~$\{t_1, t_2, t_3, t_4\}$~~

Integrated on GZoltar



http://www.gzoltar.com

Experimental Subjects

Subject	Version	Classes	Test Cases	LOCs	Coverage
JMeter	2.6	970	556	84266	34.8%
JTopas	0.8	57	160	4373	71.9%
NanoXML	2.2.3	29	9	4660	56.2%
org.jacoco.report	0.5.7	59	235	2600	97.3%
XML-Security	1.5.0	353	462	24542	64.7%

Research Question I

RQ1: Can RZOLTAR efficiently minimize the test suite, maintaining the same code coverage?

	Original	RZoltar		greedy	
Subject	$\|T\|$	$\|T_m\|$	%	$\|T_m\|$	%
JMeter	556	**237**	**57.37%**	255	54.14%
JTopas	160	**27**	**83.13%**	29	81.88%
NanoXML	9	**7**	**22.22%**	8	11.11%
org.jacoco.report	235	**63**	**73.19%**	66	71.91%
XML-Security	462	**140**	**69.70%**	167	63.85%

Research Question I

RQ1: Can RZoltar efficiently minimize the test suite, maintaining the same code coverage?

	RZoltar			greedy		
Subject	t	σ	#	t	σ	#
JMeter	**1.115**	0.027	**2**	16.190	0.076	1
JTopas	**0.475**	0.015	**2**	0.725	0.004	1
NanoXML	**0.042**	0.004	**2**	0.169	0.005	1
org.jacoco.report	**0.205**	0.004	**1**	0.679	0.007	**1**
XML-Security	**3.046**	0.066	**3**	16.852	0.034	1

Research Question II

RQ2: What is the execution time reduction of RZOLTAR's minimized test suite when compared to the original suite (and the suite computed using the greedy approach)?

	Original	RZoltar		greedy	
Subject	t	t	%	t	%
JMeter	28.844	23.405	**18.86%**	23.878	17.22%
JTopas	2744.891	852.067	68.96%	836.914	**69.51%**
NanoXML	0.417	0.361	**13.43%**	0.374	10.31%
org.jacoco.report	3.423	1.206	**64.77%**	1.627	52.47%
XML-Security	30.056	13.092	**56.44%**	18.089	39.82%

Conclusions

1. We propose a technique for test suite minimization based on constraint solving programming, which efficiently reduces the size of the test suite, maintaining full coverage.
2. The proposed technique has been implemented within the GZoltar toolset [5], more specifically in a cutting-edge Eclipse view dubbed RZoltar, this way providing an ecosystem for testing and debugging software programs.
3. We empirically evaluate the test minimization capabilities of RZ OLTAR using large, real world software programs. We observed averaged reductions of 61.17% in terms of test suite size and 63.98% of execution time reduction.
4. We compare the performance and results of our approach with greedy, known as an effective time algorithm [22].

Questions?