# JCHARMING: A Bug Reproduction Approach using Crash Traces and Directed Model Checking

Mathieu Nayrolles[1], Wahab Hamou-Lhadj[1], Sofiène Tahar[2] and Alf Larsson[3]

[1]Software Behaviour Analysis (SBA) Research Lab, ECE, Concordia, Montréal, Canada
[2]Hardware Verification Group (HVG) Research Lab, ECE, Concordia, Montréal, Canada
[3]PLF System Management, R&D Ericsson, Stockholm, Sweden

*mathieu.nayrolles@gmail.com, wahab.hamou-lhadj@concordia.ca, tahar@ece.concordia.ca, alf.larsson@ericsson.com*

March 4, 2015

# Context: Software are released with bugs.

- Despite testing and verification, softwares are pledged to be released with latent bugs.
- Latent bugs will cause field crashes / faillures.
- Patching field faillures is challenging:
  - We have to know about them.
  - Information is scarce and inconsistent
  - Most valuable information are the one that help to reproduce a bug [Bettenburg, 2008].

# Related Works: Current ways to reproduce a crash.

**Record and replay**

- Instrumentation of source code
- Record on-field execution
- Replay in-house

- Cheap & easy to implement
- Yield good results
- Overhead (1% to 1066%) and privacy concerns

- JRapture'00, BugNet'05, ReCrash'08

**In house crash reproduction**

- Core dump.
- Forward Symbolic Execution.
- Backward Symbolic Execution.

- Yield average results.
- NP-Compex Problem
- Exponential learning curve.
- Privacy concerns.

- BugRedux'12, RECORE'13, STAR'13

# JCHARMING: A different direction

- Avoid code instrumentation
  - 0% overhead
- Do no yield privacy concerns
  - Scalable to real-world and industrial/proprietary software systems
- Leverage the stack traces resulting from a crash
  - More and more often present in bug reports
- JCHARMING uses directed model checking and backward slicing.

# Prelimenaries: Model Checking

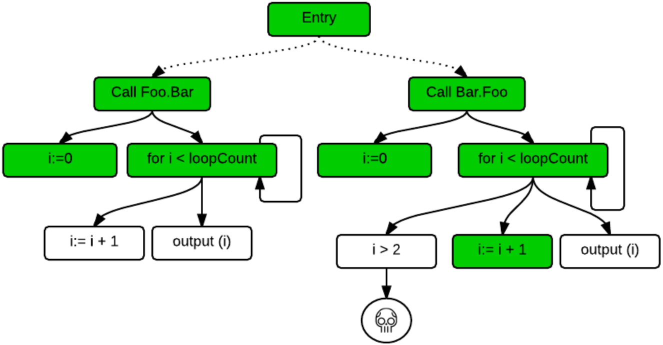
- Checks if a given system under test (SUT) meets a specification $p$ by exhaustively testing every states. [Visser, 2003], [Kropf, 1999]
- Generates a counter example if $p$ cannot be met.
- Represented by a Kripke structure [Kripke, 1963]:

$$SUT =< S, T, P > \text{ , } (SUT, x) \models p$$

- Ensures that $p$ is reached at some point and not that $p$ holds nor $\forall x, p$ is satisfiable.
- In JCHARMING, we aim to verify that $\forall$ states the program does not crash:

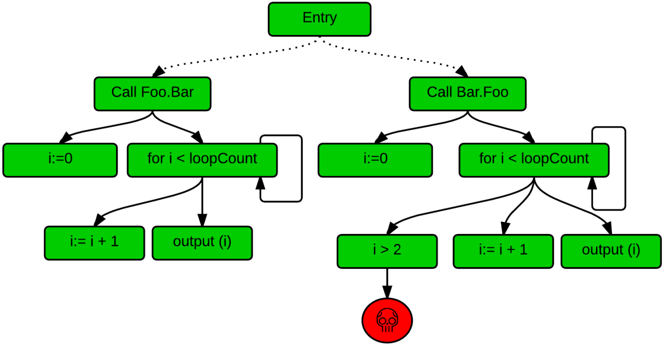
$$\forall x.(SUT, x) \models \neg c$$

# Prelimenaries: **Testing**, Model Checking and Directed Model Checking



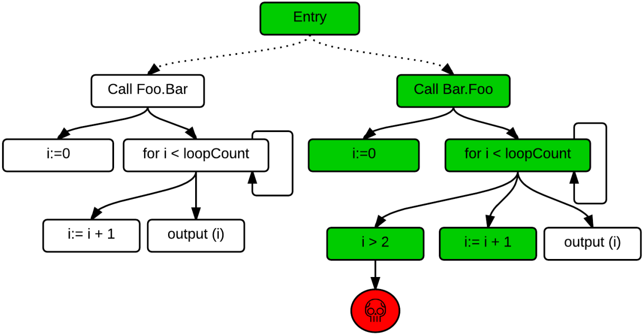

- Depends on the tester understanding of the SUT.
- Ineficient because it is not exhaustive.

# Prelimenaries: Testing, **Model Checking** and Directed Model Checking

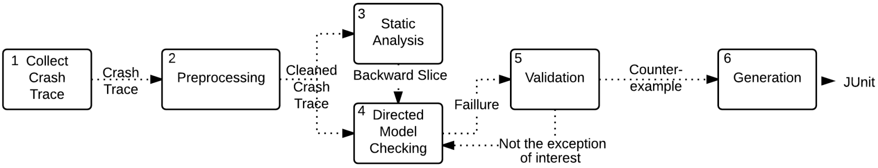



- Explores each and every state of the program, hence it is complete.
- Impractical for real-world and large systems

# Prelimenaries: Testing, Model Checking and **Directed Model Checking**




- Explores only the states that may lead to a specific location. [Rungta, 2009]
- Use insights – generally heuristics – about the SUT to prune states.




- **Step 1: Collect the crash trace**

```
1.javax.activity.IAE:loopTimes should be < 3
2.   at Foo.bar(Foo.java:10)
3.   at GUI.buttonActionPerformed(GUI.java:88)
4.   at GUI.access$0(GUI.java:85)
5.   at GUI$1.actionPerformed(GUI.java:57)
6.   caused by java.lang.IndexOutOfBoundsException :  3
7.   at saner.Foo.buggy(Foo.java:17)
8.   and 4 more ...
```

- **Step 2: Preprocessing**

# Step 3: Building the Backward Static Slice

- Large systems does not necessary contain all the methods that have been executed starting from the entry point of the program to the crash [Oracle, 2011]
- We need to complete the missing frames of the stack traces
- A backward slice contains all possible branches that may lead to a point $n$ from a point $m$ as well as the definition of the variables that control these branches
- We perform a static backward slice between each frame to compensate for possible missing information in the crash trace:

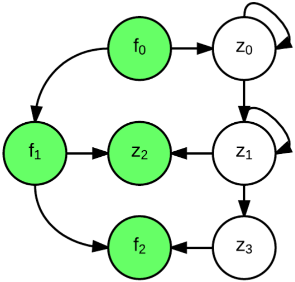
$$bslice_{[entry \leftarrow f_0]} = bslice_{[f_1 \leftarrow f_0]} \cup \ldots \cup bslice_{[entry \leftarrow f_n]}$$

# Step 3: Building the Backward Static Slice (Cont'd)

- The union of the sub backward static slices is a subset of the backward static slice from $f_0$ to entry.

  $$\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq bslice_{[f_{entry} \leftarrow f_0]}$$

- If $z_2$ is a prerequisite to $f_2$:
  - $bslice_{[f_{entry} \leftarrow f_0]} = \{f_0, f_1, f_2, z_0, z_1, z_2, z_3\}$
  - $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} = \{f_0, f_1, f_2, z_2\}$



# Step 3: Building the Backward Static Slice (Cont'd)

- The search space for the model checker is limited by the backward slice:

$$\exists x. \begin{pmatrix} \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT \\ x.\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT \end{pmatrix} \models c_{i>2}$$

- It exists a sequence of states transitions $x$ that satisfies $c_{i>2}$ in $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]}$

# Step 4: Directed Model Checking

- We use Java PathFinder (JPF)
  - JVM for Java bytecode verification.
  - Front-end for the SPIN model checker.
  - Developped and maintained by NASA.
- Generate states
- Forward
  - Generates the next state $S_{t+1}$ and add it to the backtrack table
- Backward
- Backtrack
  - Restore the last state of the backtrack table.
- Restore state
- Check properties
  - Is triggered after each forward, backward and restore operations.

# Step 4: Directed Model Checking (Cont'd)

- We modified the *generate states* and the *forward steps*.
- The *generate states* is populated with:

$$\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT$$

- The *foward step* can explore a state if $s_{i+1}$ and the transition $x$ from $s_i$ to $s_{i+1}$ are in

$$\begin{pmatrix} \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT \\ x.\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT \end{pmatrix}$$

- JPF is now directed and explores only a sub-system of the SUT.

# Step 5: Verification

- We modify the *check properties* step of JPF:
  - If the current states transitions $x$ yield an exception
  - We execute $x$ and compare the stack trace to the original
  - If the two exceptions match, the bug is *reproduced*.
- Bug can be partially reproduced if the generated exception matches the original by a factor of $t$:
  - Same faillure can be reached by different paths. [Kim, 2013]
  - Might enhance the comprehension.
  - Might speed-up the deployment of a fix.

# Step 6: Generating Test Cases for Bug Reproduction

- A JUnit test suite allows developers to reproduce the bug on the press of a button
- JPF keeps track of the visited states during the model checking process.
- We leverage this ability to create the required objects and call the methods leading to the crash.

# Experiments

- Aim to answer the following RQ: *Can we use crash traces and directed model checking to reproduce on-field bugs in a reasonable amount of time?*
- Randomly selected 1 from 10 bugs containing a stack trace for each system

| SUT | KLOC | NoC | Bug #ID |
|---|---|---|---|
| Ant | 265 | 1233 | 38622, 41422 |
| ArgoUML | 58 | 1922 | 2603, 2558, 311, 1786 |
| dnsjava | 33 | 182 | 38 |
| jfreechart | 310 | 990 | 434, 664, 916 |
| Log4j | 70 | 363 | 11570, 40212, 41186, 45335, 46271, 47912, 47957 |
| MCT | 203 | 1267 | 440ed48 |
| pdfbox | 201 | 957 | 1412, 1359 |

# Experiments (Cont'd)

- 85% success ratio (17/20) with $t = 80\%$
- 16 minutes average time to reproduce.

| SUT | Bug #ID |
|---|---|
| Ant | **38622 (25.4m)**, 41422 (-) |
| ArgoUML | 2603 (9.4m), 2558 (10.6m), **311 (11.3m)** , 1786 (9.9m) |
| dnsjava | **38 (4m)** |
| jfreechart | **434 (27.3m)**, 664 (31.2m) , **916 (26.4m)** |
| Log4j | **11570 (12.1m)** , **40212 (15.8m)**, 41186 (16.7m), 45335 (-) ,**46271 (13.9m)** , **47912 (12.3m)** , 47957 (-) |
| MCT | **440ed48 (18.6m)** |
| pdfbox | 1412 (19.7m) , 1359 (-) |

- JCHARMING uses model checking directed by backward static slice and is able to reproduce bug in a reasonable amount of time.

# Experiments: Reproduced example

### Argo UML #311

*I open my first project (Untitled Model by default). I choose to draw a Class Diagram. I add a class to the diagram. The class name appears in the left browser panel. I can select the class by clicking on its name. I add an instance variable to the class. The attribute name appears in the left browser panel. I can't select the attribute by clicking on its name. Exception occurred during event dispatching:*

```
1.  java.lang.NullPointerException:  2.  at
3.  uci.uml.ui.props.PropPanelAttribute
...
28.  at java.awt.EventDispatchThread.pumpEvents (EventDispatch
Thread.java:90)
29.  at java.awt.EventDispatchThread.run(EventDispatch
Thread.java:82)
```

# Experiments: Reproduced example (Cont'd)

```
try {

    org.argouml.ui.ProjectBrowser v0 = new org.argouml.ui.ProjectBrows
    v0.setNavigatorPaneVisible(true);
    org.argouml.ui.NavigatorPane v1 = v0.getNavPane();
    org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram v2 =
            new org.argouml.uml.diagram.static_structure.ui.UMLClassDi
    java.awt.event.ActionEvent v3 = new java.awt.event.ActionEvent((Ob
    v2.dispatchEvent();
    org.argouml.uml.ui.ActionAddAttribute v4 = new org.argouml.uml.ui.
    java.awt.event.ActionEvent v5 = new java.awt.event.ActionEvent((Ob
    v4.dispatchEvent();
    v0.setSelection(v4, null);

} catch (Exception e) {
     differences = 0;
     StringTokenizer tokenizerFaillure = new
     StringTokenizer(e.getStackTrace()
     .toString(), "\n");
     while (tokenizeOriginalFaillure.hasMoreTokens()) {
         if (tokenizeOriginalFaillure.nextToken().compareTo(
                 tokenizerFaillure.nextToken()) != 0)
         {
             differences++;
         }
     }
```

# Experiments: Partially Reproduced Example

**Jfreechart #664**

*In ChartPanel.mouseMoved there's a line of code which creates a new ChartMouseEvent using as first parameter the object returned by getChart(). For getChart() is legal to return null if the chart is null, but ChartMouseEvent's constructor calls the parent constructor which throws an IllegalArgumentException if the object passed in is null*

```
1.  java.lang.IllegalArgumentException:  null source
2.  at java.util.EventObject.<init>( EventObject.java:38)
3.  at
4 org.jfree.chart.ChartMouseEvent.<init>
(ChartMouseEvent.java:83)
5.  at org.jfree.chart.ChartPanel
.mouseMoved(ChartPanel.java:1692)
6.  <deleted entry>
```

# Experiments: Not Reproduced Example

**Log4j #47957**

*Configure SyslogAppender with a Layout class that does not exist; it throws a NullPointerException. Following is the exception trace:*

```
 1.  10052009 01:36:46 ERROR [Default:  1]
struts.CPExceptionHandler.execute
RID[(null;25KbxlK0voima4h00ZLBQFC;236Al8E60000045C3A
7D74272C4B4A61)]
2.  Wrapping Exception in ModuleException
3.  java.lang.NullPointerException
4.  at org.apache.log4j.net.SyslogAppender
.append(SyslogAppender.java:250)
5.  at org.apache.log4j.AppenderSkeleton
.doAppend(AppenderSkeleton.java:230)
```

# Conclusion

- JCHARMING (Java CrasH Automatic Reproduction by directed Model checking)
- Automatic bug reproduction technique that combines crash traces and directed model checking
- Direct the model checking engine with a backward static slice
- Was able to reproduce fully or partially 85% (17/20) of the bugs

- Stress JCHARMING with more bugs
  - Fine tune the approach
  - Assess the scalability on larger / proprietary systems
- Test the performances of JCHARMING with multi-threading related bugs

# QUESTIONS?