Wahab Hamou-Lhadj

Associate Professor
Department of Electrical and Computer Engineering
Concordia University
abdelw@ece.concordia.ca

May 2, 2016

The Mission of the Software Behaviour Analysis Research Lab

To investigate techniques and tools **to help software (and system) analysts understand and analyze the behaviour of complex software systems** with the primary goal of enhancing maintenance, security, and addressing performance problems

Topics of interest

- Tracing, logging, and runtime monitoring
- Big data analytics of log and trace data
- Log management and engineering
- Cloud-based logging and tracing infrastructures
- System diagnosis and observability
- Dynamic analysis for program comprehension
- Model-driven software tracing and debugging
- Online system observation and surveillance
- Software healing and adaptation
- Visualization and classification of program behaviour

Current Projects

- AHLS - Advanced Host-Level Surveillance
- D2K - From Data 2 Knowledge for Better System Maintenance
- OpenSim - An Open Architecture for Aircraft Simulation Integration and Monitoring Methods Using the HLA Standard
- AVIO 508 - Diagnostics for Real Time Distributed Multi-core Architecture in Avionics

HQP

Neda Ebrahimi (PhD)

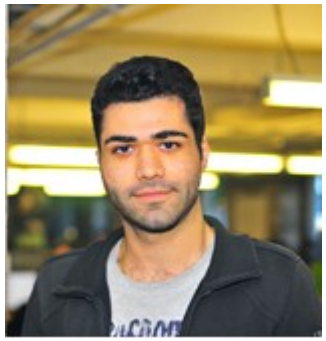
Korosh K. Sabor (PhD)

Mathieu Nayrolles (PhD)

Ava Khanmohammadi (PhD)

Md Islam Shariful (PhD)

Mohammad R. Rejali (PhD)



Amir Gahroosi (RA)

Sama Khosravifar (MSc.)

Abhishek Koyalkar (MEng.)

A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces

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

May 2nd, 2016

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.

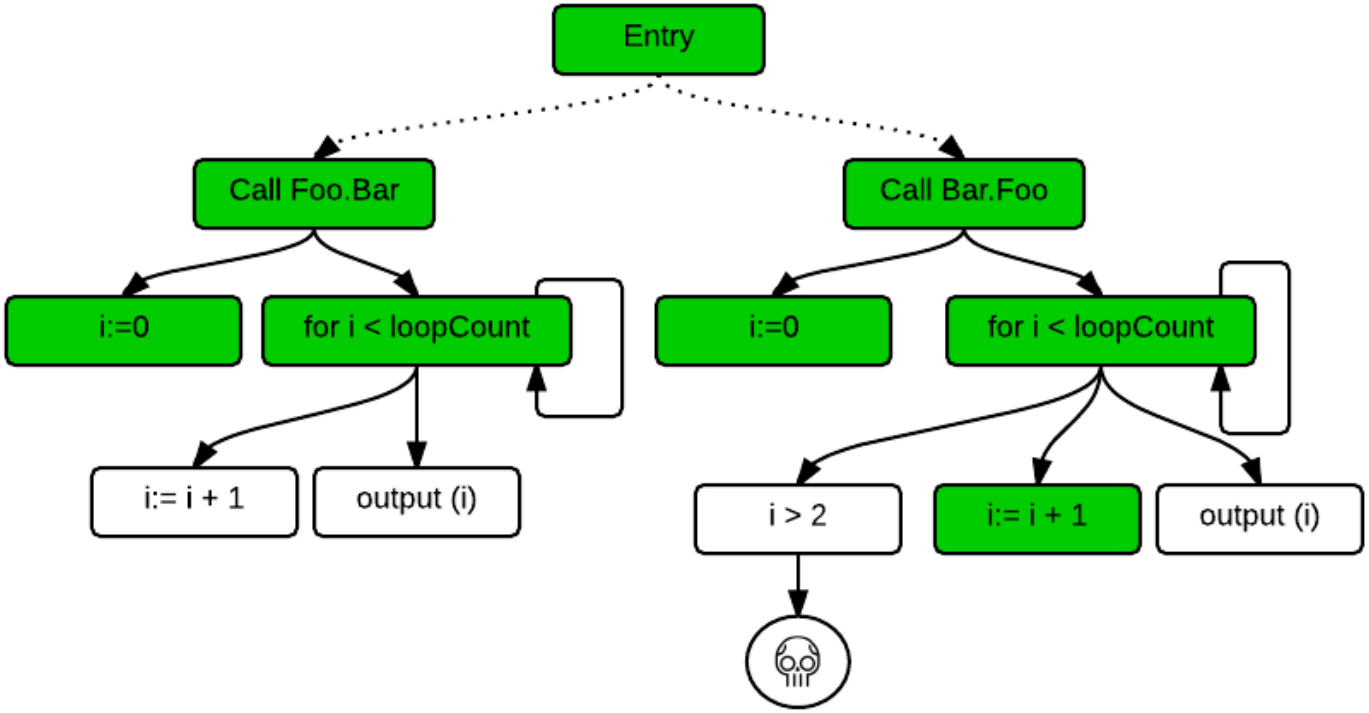
- Yield average results.
- NP-Compex Problem
- Exponential learning curve.
- Privacy concerns.

- 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.

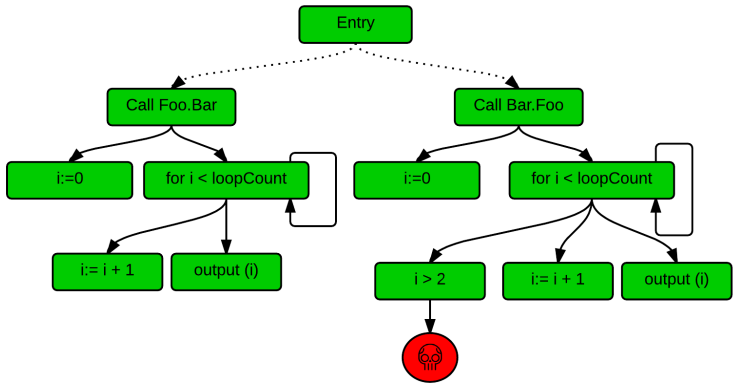
Prelimenaries: **Testing**, Model Checking and Directed Model Checking



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

Prelimenaries: Model Checking

- Checks if a given system under test (SUT) meets a specification p by exhaustively testing every states. [Visser, 2003], [Kropf, 1999]

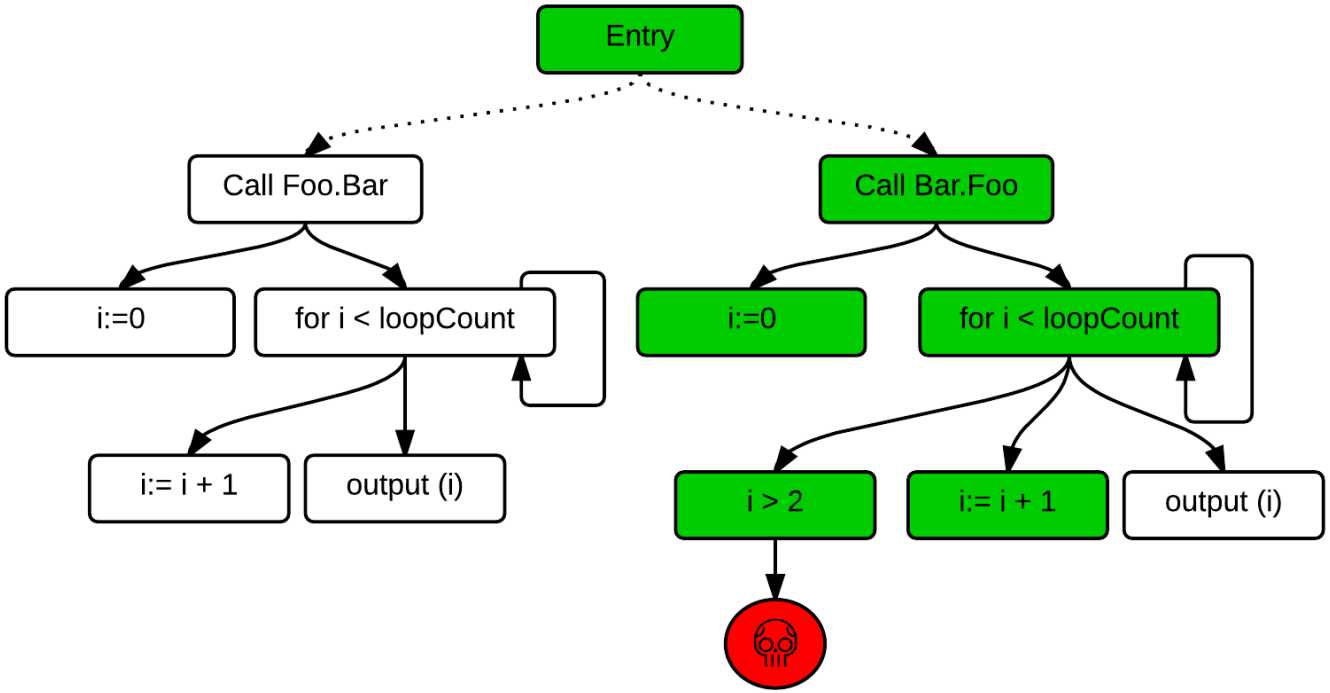


- Ensures that p is reached at some point and not that p holds nor $\forall x, p$ is satisfiable.
- We aim to verify that $\forall$ states the program does not crash:

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

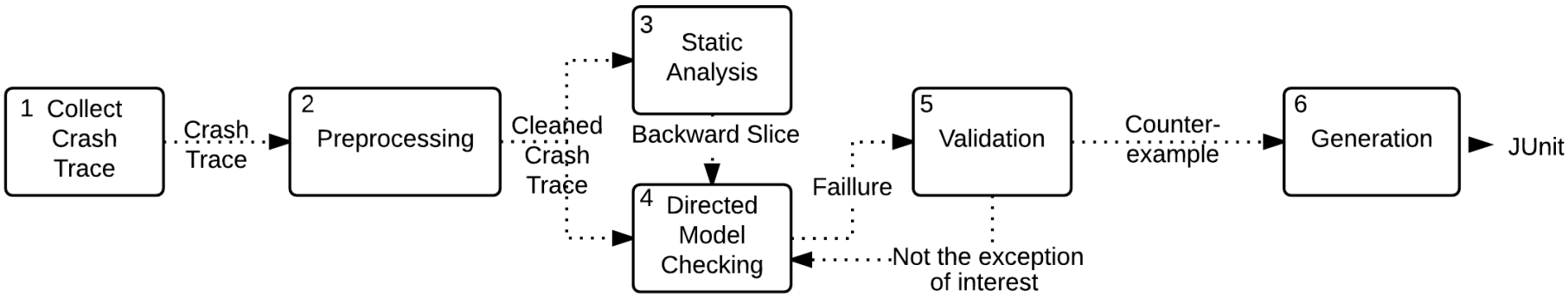
- Explores each and every state of the program, hence it is complete.
- Impractical for real-world and large systems

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.

The JCHARMING approach



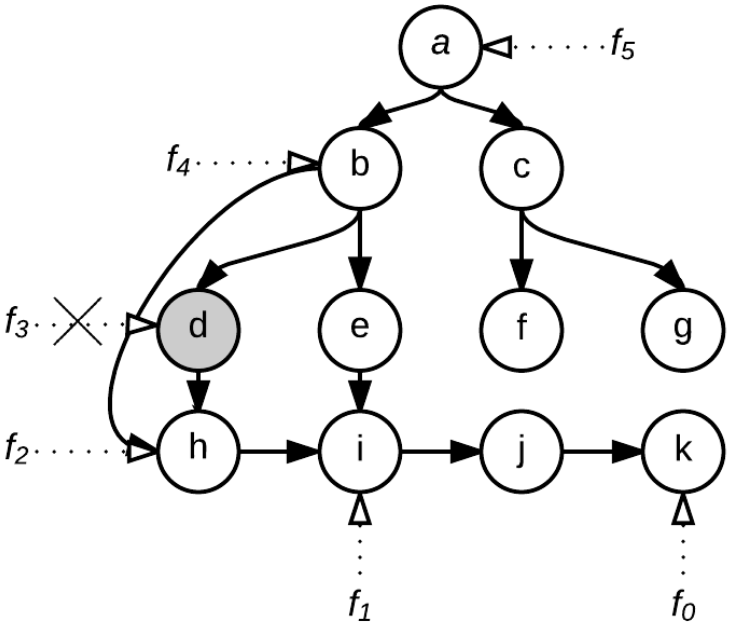
- **Step 1: Collect the crash trace**

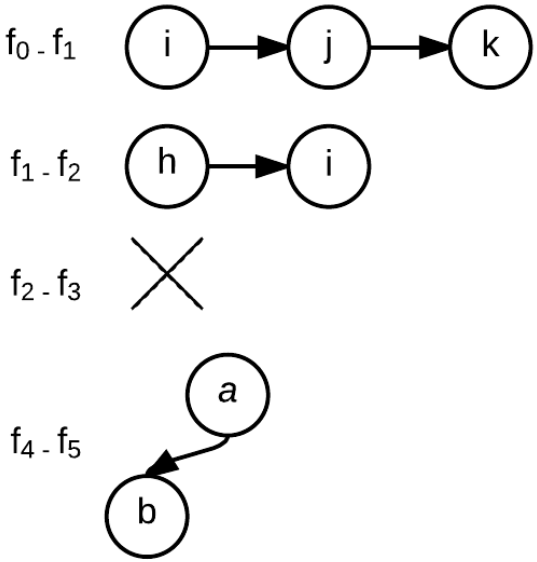
```
1.javax.activity.IAE:loopTimes should be < 3
2.   at Foo.bar(Foo.java:10)
3.   at GUI.buttonActionPerformed(GUI.java:88)
4.   at GUI.access$0(GUI.java:85)
5.   at GUI$1.actionPerformed(GUI.java:57)
6.   caused by java.lang.IndexOutOfBoundsException :  3
7.   at saner.Foo.buggy(Foo.java:17)
8.   and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

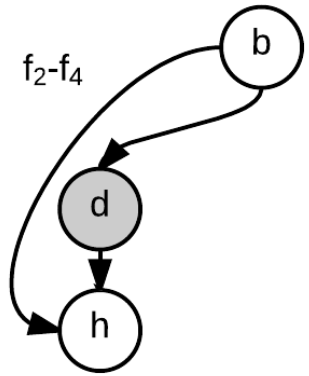
- **Step 2: Preprocessing**

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

- 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
- Let's assume we have $T = \{k, i, f, d, b, c\}$
- Without using T the backward slice is $\{a, b, d, e, h, u, j, k\}$.
- With T, and frame by frame it is $\{a, b, d, h, i, j, k\}$.







Step 4: Directed Model Checking

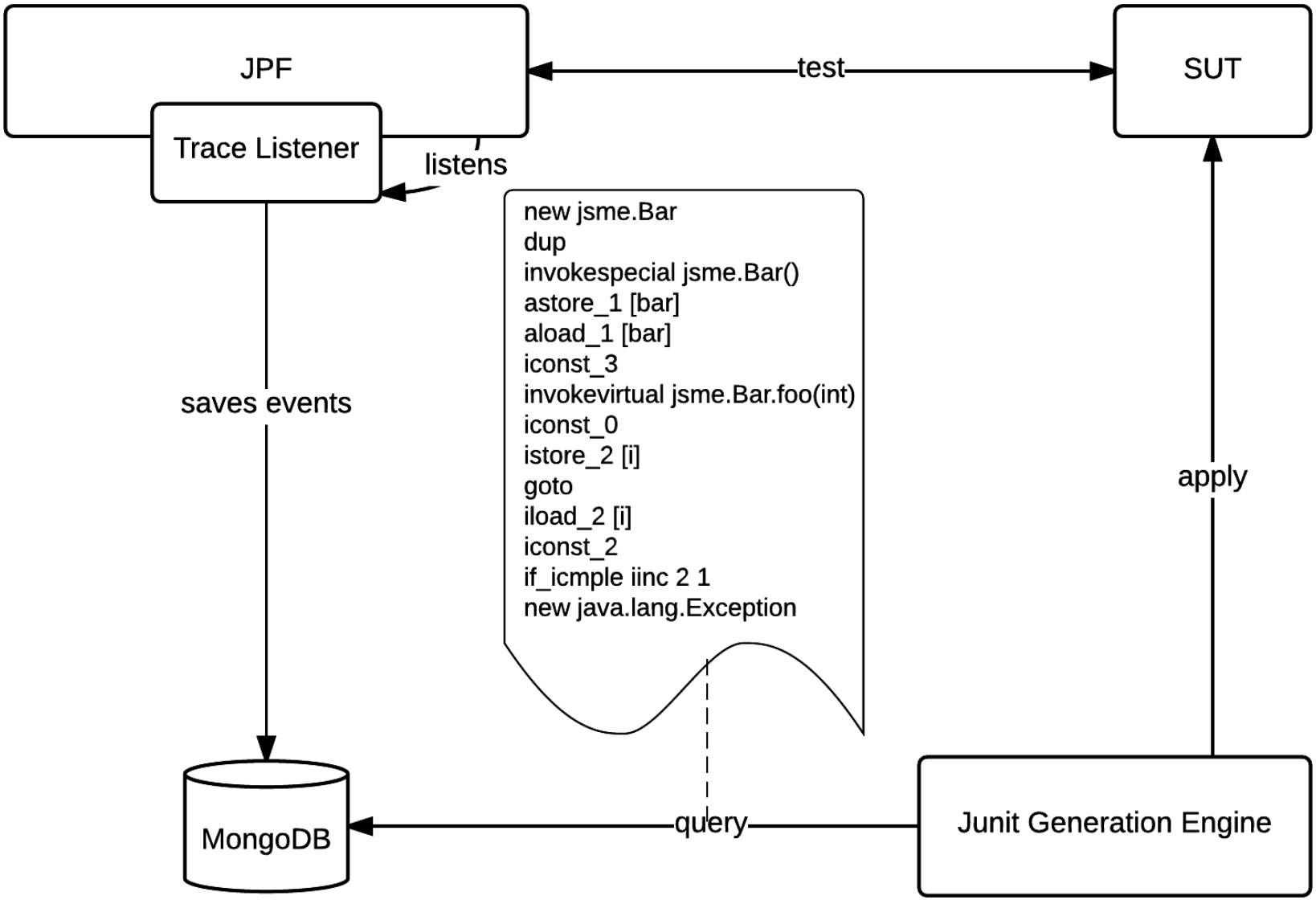
- We use Java PathFinder (JPF)
  - JVM for Java bytecode verification.
  - Front-end for the SPIN model checker.
  - Developped and maintained by NASA.
- Generate states
- Forward
  - Generates the next state S_{t+1} and add it to the backtrack table
- Backward
- Backtrack
  - Restore the last state 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 $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT$
- 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*.

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

Step 6: Generating Test Cases for Bug Reproduction



Experiments And Conclusion

- 30 bugs belonging to 10 open sources systems (Ant, ArgoUML, DnsJava, jFreeChart, Log4j, MCT, PDFBox, Mahout, Hadoop, ActiveMQ).
- 80% success ratio (24/30).
- Average success time is 19 minutes.
- Average fail time is 11 minutes.
- Average JUNIT length is 5 java statements.

- Stress JCHARMING with more bugs
- Test the performances of JCHARMING with multi-threading related bugs

QUESTIONS?