Detecting deadlocks using static analysis in .NET

Filip Navara

filip.navara@gmail.com

Week 1: What did I do?

- Collected papers that caught my interest and may provide valuable information for further study
  - Published at http://goo.gl/jSudL
  - To be considered: Move to SVN ?
- Created code repository for experimental work
  - Accessible at https://svn.assembla.com/svn/nodeadlock/
- Experiment with extracting thread entrypoints from larger .NET program

References: Exception handling in control-flow graph

- Analysis and Testing of Programs With Exception-Handling Constructs, Saurabh Sinha and Mary Jean Harrold
- Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow, Jang-Wu Jo and Byeong-Mo Chang
- *May be possible to decouple the exception handling from normal flow for the purpose of our analysis and thus make the function control flow graphs smaller and easier to walk through*

References: .NET Platform

- ECMA 335: Common Language Infrastructure (CLI)
  - Byte code specification, type system, overload resolution, ...
- Threading in C#, Joseph Albahari
  - Description of locking primitives in the .NET framework

References: Static analysis in .NET [MoonWalker, csLint]

- MMC: the Mono Model Checker, Theo et al.
- MoonWalker: verification of .NET programs, Neils et al.
- Software Model Checking for Mono, Niels
- Optimising Techniques for Model Checkers, Viet
- Memoised Garbage Collection for Software Model Checking
- csLint: http://www.garret.ru/csharp.html

References: Deadlock detection using static analysis

- Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring, Rahul et al.
- Effective Static Deadlock Detection, Mayur et al.
- Potential Deadlock Detection (GoodLock), Havelund et al.
- Static Deadlock Detection for Java Libraries, Williams et al.

References: Uncategorized (so far)

- Extracting a Petri Net Representation of Java Concurrency, Bateman and Pouarz
- Variably Interprocedural Program Analysis for Runtime Error Detection
- Practical Virtual Method Call Resolution for Java
  - *Useful for reducing call graphs and thus also reducing search space and improving accuracy at expense of additional time needed for determining the reduced call graph*
  - *Techniques presented in this paper could possibly be applied to resolve C# delegate call trees*

Problem: C# delegates

-
  ```
  event EventHandler Click;

  if (Click != null)
    Click(this, EventArgs.Empty);

  button1.Click += button1_Click;

  public void button1_Click(object sender,
    EventArgs e)
  {
    ...
  }
  ```
- When delegate is called, which method could be invoked?

Problem: C# delegates, contd.

- Naive approach
  - For each delegate type T (eg. EventHandler, Action, Action<T>, etc.):
    - Detect all invocations of new T(*m*)
    - Consider all invocations of T() to call all possible methods *m*
  - Pro: Easy to implement, low memory footprint
  - Con: Adds plenty of edges to call graph that cannot happen at run-time and thus are likely to cause false positives

Problem: C# delegates, contd.

- Graph approach
  - Contruct a graph
    - Each node represents a delegate variable (static, field, local or parameter [stack location?]) and there are nodes for all delegate variables in the analyzed program
    - Each edge represents an assignment to variable in the program, either implicit (a = b) or explicit (eg. method call created edges between local variables and parameters)
    - Virtual method invocations have to be resolved using overload resolution [?]
  - Each invocation of delegate variable could be traced back by DFS to all possible assigned values
  - Pros: Low false positive rates, sparser call graph
  - Cons: Memory footprint [?], analysis time [?]
  - How to handle arrays? What about aliasing?

Week 1: What do I plan to do?

- Create test cases that exercise various edge cases in call graph extraction
  - Calling using delegates, events
  - Virtual method calls
  - Thread static variables
- Read the „Effective Static Deadlock Detection“ paper and presentation by Mayur et al.