# Statically Verifying API Usage Rule using Tracematches

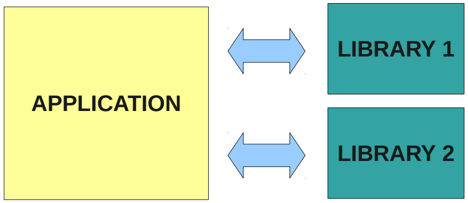
Xavier Noumbissi, Patrick Lam

University of Waterloo

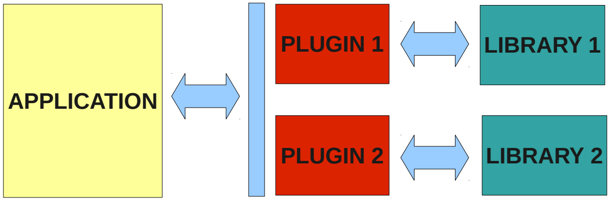
November 4, 2010

## Outline

## Software relies on Libraries



## Plug-in / Wrapper-Library



## Verification Properties

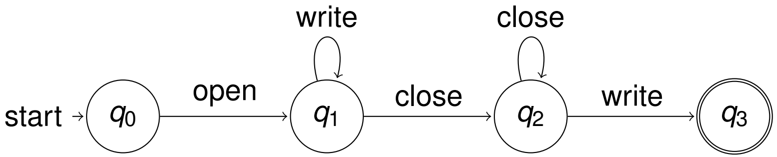
Two types of properties:

- State Properties: Pre- and Post-Conditions
- Temporal Properties: Typestates, LTL, etc.

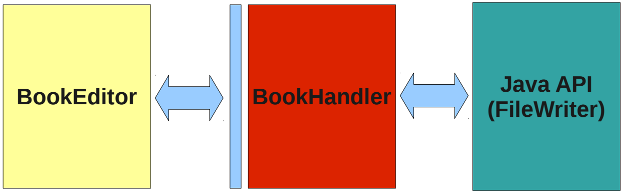
## Tracematches: Unauthorized events sequence

Example: Do not write to a file after closing it.

Regular Expression: `open write* close+ write`



## Example



## Application & Plug-in

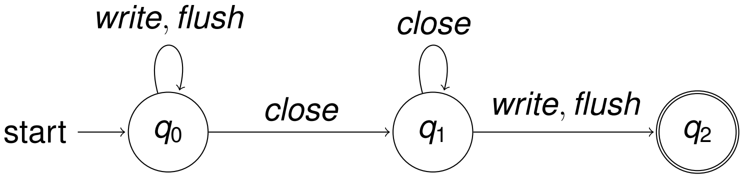
```
public class BookHandler {
    FileWriter l1;
    public void openBook(String b){
        l1 = new FileWriter(b);
    }

    public void printChapter(String s){
        l1.write(s);
    }

    public void printSection(String s){
        if (null == s) l1.flush();
        else l1.write(s);
    }

    public void addIndex(String[] t){
        for(int k=0; k<t.length; ++k) {
            l1.write(t[k]);
        }
        FileWriter l3 = l1; //Illustrates
        l3.close();         // aliasing
    }

    public void close(){
        l1.flush();
        l1.close();
    }
}
```

```
public class BookEditor {
  public static void main(String [] args){
    BookHandler b = new BookHandler();
    String []index = {"book", "chapter"};
    StringBuilder str = new StringBuilder();

    b.openBook("book.txt");
    b.printChapter("A chapter");

    str.append("Not in abstract CFG");

    b.printSection("A section");

    b.addIndex(index);

    b.close();
  }
}
```

ACFG

## A Usage Rule for `java.io.FileWriter`

No `write` or `flush` on a `FileWriter` after a `close`.



## A Violation of the Rule

The developer expects:

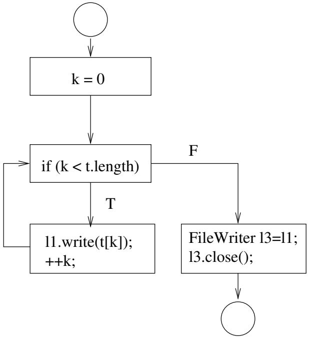
- `close`: closes the book's stream
- `addIndex`: does not close the book's stream

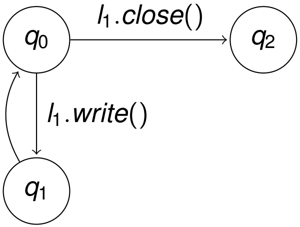
But:

- `addIndex`: closes the stream
- `close`: flushes the stream before closing it

## Method summaries

Summary of `BookHandler.addIndex`





Must-alias information at intraprocedural level: `l1` and `l3` must alias

## Plug-in summary

- Reusable as long as Plug-in code unchanged
- Represents all events sequences on `FileWriter` objects
- Storage of summaries improves scalability

## Abstracted CFG for BookEditor.main

▸ The code

- Here's a CFG which only includes the `BookHandler` calls.
- Statements on edges

$$\begin{tikzcd}
q_0 \arrow[d, "b.openBook(...)"] \\
q_1 \arrow[d, "b.printChapter(...)"] \\
q_2 \arrow[d, "b.printSection(...)"] \\
q_3 \arrow[d, "b.addIndex(...)"] \\
q_4 \arrow[d, "b.closeBook(...)"] \\
q_6
\end{tikzcd}$$

## Edge Substitution - 1

Substitute method-call edges with NFA summaries:

- Summarizes application behavior with respect to tracematch events
- May-alias information at interprocedural level
- Unless `FileWriter` objects may not alias

## Edge Substitution - 2

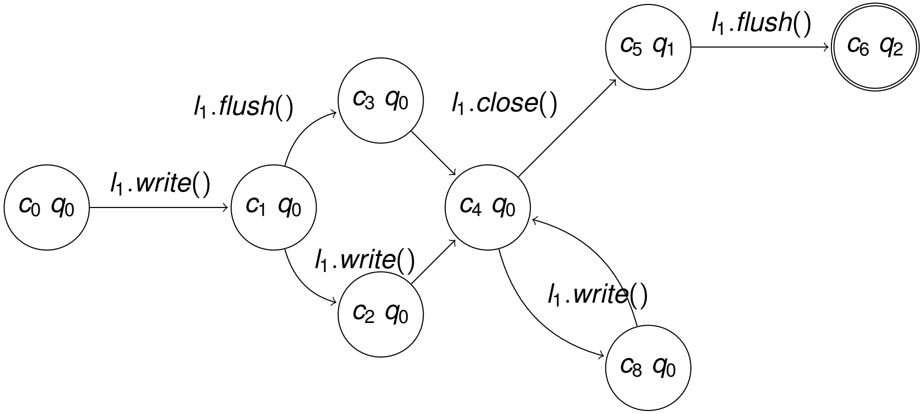
$$\begin{tikzcd}
 & & c_2 \arrow[dr, "l_1.close()"] & & c_5 \arrow[r, "l_1.flush()"] & c_6 \arrow[r, "l_1.close()"] & c_7 \\
c_0 \arrow[r, "l_1.write()"] & c_1 \arrow[ur, bend left, "l_1.write()"] \arrow[dr, bend right, "l_1.flush()"'] & & c_4 \arrow[ur] \arrow[dr, bend right] & & & \\
 & & c_3 \arrow[ur] & & c_8 \arrow[ul, bend right, "l_1.write()"'] & &
\end{tikzcd}$$

## Tracematch: Runtime Monitor (RM)

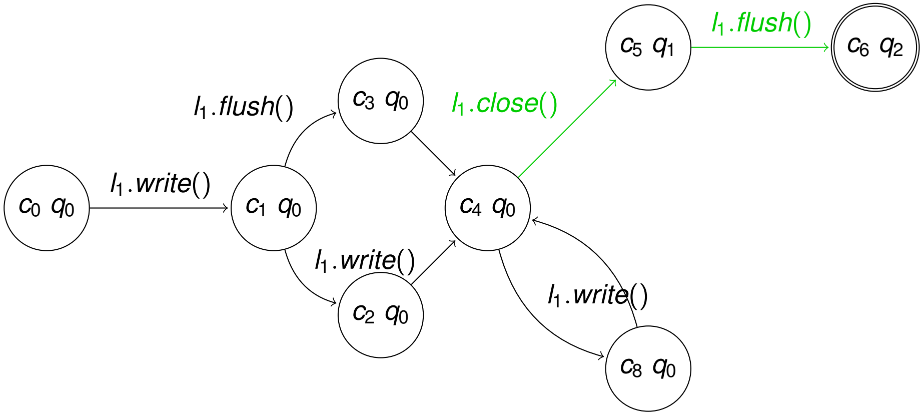
Combined execution of application and RM:

- *Synchronous* Product Automaton (SPA): defines a transition for an event only if it occurs in both automatons
- Shared actions: tracematch events

## Synchronous Product Automaton - 1



## Synchronous Product Automaton - 2



An accepting path represents a violation of the rules.

## Our approach: Four Phases

1. Compute plug-in summaries (Preliminary phase)
2. Generate application Abstract CFGs
3. Integrate plug-in summaries to abstract CFGs
4. Build SPA and check for accepting paths

## Implementation

- Uses the Soot Framework
- Interaction of inter- and intra-procedural analysis
- Intraprocedural analysis as a region-based analysis

## Computing Plug-in Summaries

Advantages of storing summaries:

- Effects of whole program analysis with partial code
- No need to have plug-in code
- Manual creation of plug-in summary
- Plug-in summary as a contract

## Library Object Accesses

Library objects may be accessible as:

- Member variables
- Local instantiated variables
- Others: out of scope

## Related Work

- **Static optimization of runtime monitors**: static verification of tracematches at the intraprocedural level (Bodden et al.)
- **Analysis of multiple interactive objects**: Uses tracematches to verify correct interaction of several objects (Naeem et al.)
- **Component Level data-flow Analysis (CLA)**: computes summaries and properties of software with partial information (source code or summaries) of its components. (Rountev et al.)

## Future Work

- Complete implementation
- Test analysis on production software
- Use of constraint-solvers to remove false positives from summaries
- Try to improve scalability of other analyses on tracematches using summaries

## Contributions

- Check that abstraction layers do not introduce bugs
- Use of summaries as contracts
- Use of summaries for other analyses on tracematches

## THANK YOU !

**Comments & Questions**