Terracotta: Mining Temporal API Rules from Imperfect Traces

Jinlin Yang
University of Virginia
jinlin@cs.virginia.edu

NJPLS 11/18/2005

Collaborated with David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das

Overview

- Problem: unavailability of specification is a big issue in defect detection

- Solution: automatically inferring specification from execution traces

- Results: better understanding of legacy code and finding more defects
  - Experiments on Windows kernel APIs and JBoss
  - Interesting Windows kernel API rules that should be checked
  - Many previously unknown bugs in Windows
  - Inferred behaviors of JBoss that are consistent with J2EE spec

Outline

- Introduction
- My approach
- Preliminary experiment on Windows kernel APIs
- Refinement of inference with new experimental results
- Contributions, future work, and conclusion

Problem

- Defect detection techniques require specifications
- Generic properties
  - E.g. absence of null-pointer dereference
  - PREfix [Bush+, SP&E00], PREfast, etc.
  - Very effective
- Application specific properties
  - E.g. lock/unlock, resource creation/deletion
  - LCLint [Evans, PLDI96], SLAM [Ball+, SPIN01], Vault [DeLine+, PLDI01], Type Qualifiers [Foster+, PLDI02], ESP [Das+, PLDI02], ESC/JAVA [Flanagan+, PLDI02], FindBugs [Hovemeyer+, OOPSLA04], Spec# [Barnett+, CASSIS04]
- Such properties are rarely available

Temporal Properties

- Example: Lock::Acq -> Lock::Rel
- Why are they important?
  - Essential for correctness
- Applications
  - Developers do care
    - *what sequence of functions should I call to access this resource?*
    - *After calling function A, what other functions must (not) I call?*
  - Can be used to verify programs
- Rarely available, hard to get right [Holzmann, FSE02]
- How do we get such temporal properties?

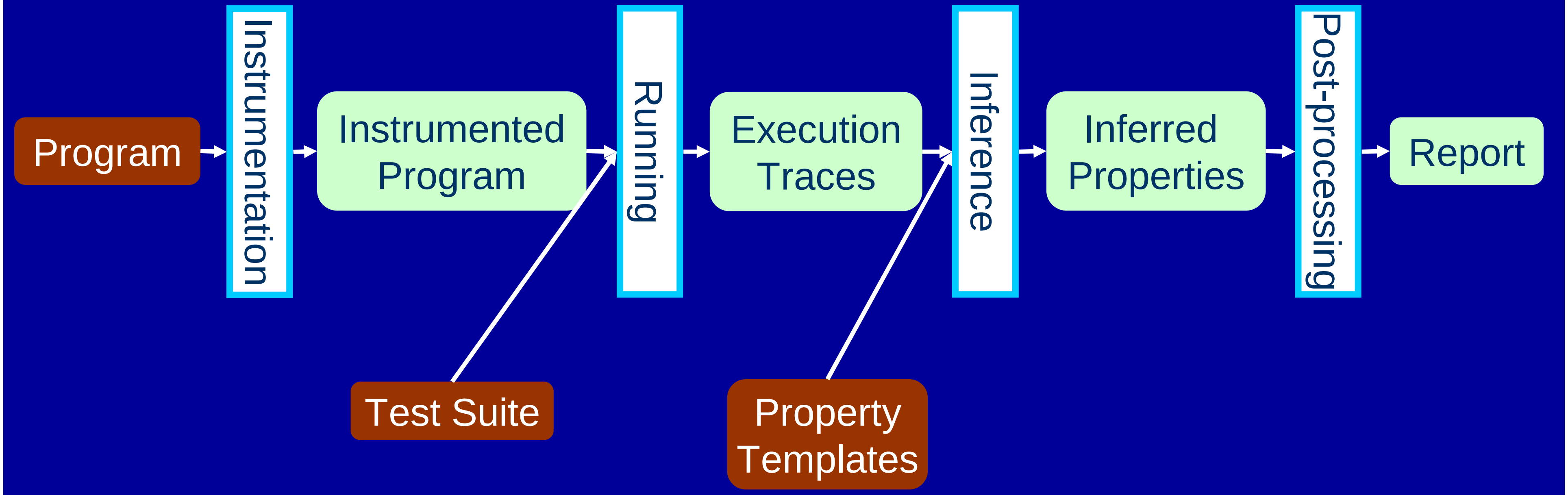
Contributions

- A novel statistical approach for inferring temporal properties from execution traces
- Combining automatic inference and verification together by feeding inferred properties to ESP
- Demonstration of the usefulness and effectiveness of this approach in realistic systems

Outline

- Introduction
- **My approach**
- Preliminary experiment on Windows kernel APIs
- Refinement of inference with new experimental results
- Contributions, future work, and conclusion

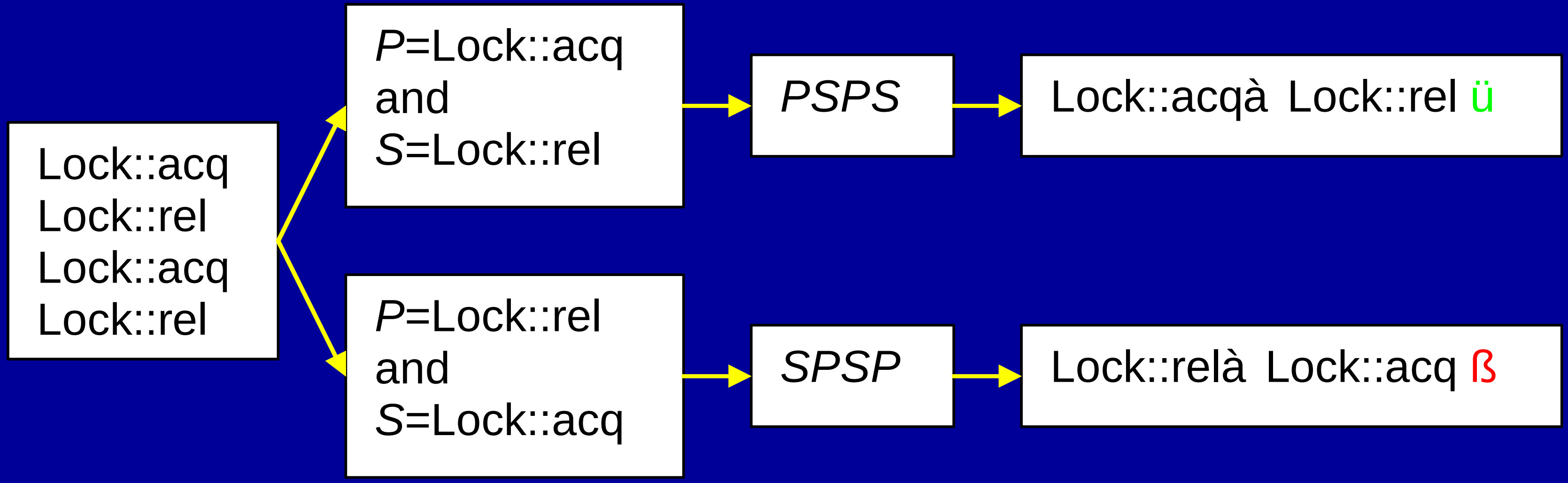
My Approach



Jinlin Yang and David Evans. Dynamically inferring temporal properties. *PASTE '04*.

An Example

- Alternating template
  *(PS)**, *P!=S*.   *P* and *S* are parameters



Lock::acq
Lock::rel
Lock::acq
Lock::rel

P=Lock::acq
and
S=Lock::rel

PSPS

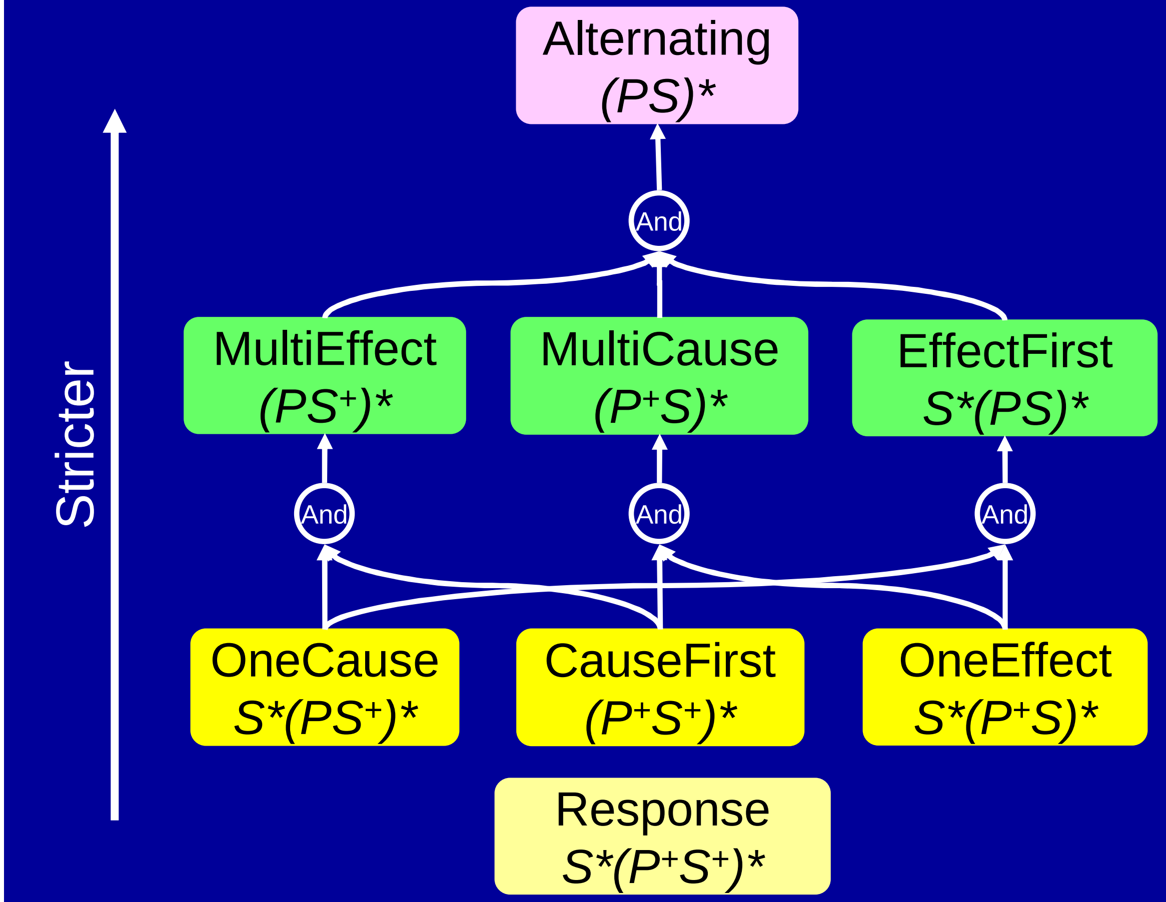
Lock::acqà Lock::rel ü

P=Lock::rel
and
S=Lock::acq

SPSP

Lock::relà Lock::acq ß

Property Templates



Alternating
$(PS)^*$

And

MultiEffect
$(PS^+)^*$

MultiCause
$(P^+S)^*$

EffectFirst
$S^*(PS)^*$

And

And

And

OneCause
$S^*(PS^+)^*$

CauseFirst
$(P^+S^+)^*$

OneEffect
$S^*(P^+S)^*$

Response
$S^*(P^+S^+)^*$

Stricter

- For each pair of two events
- Decide if they satisfy CauseFirst, OneCause, or OneEffect
- Derive the strictest pattern

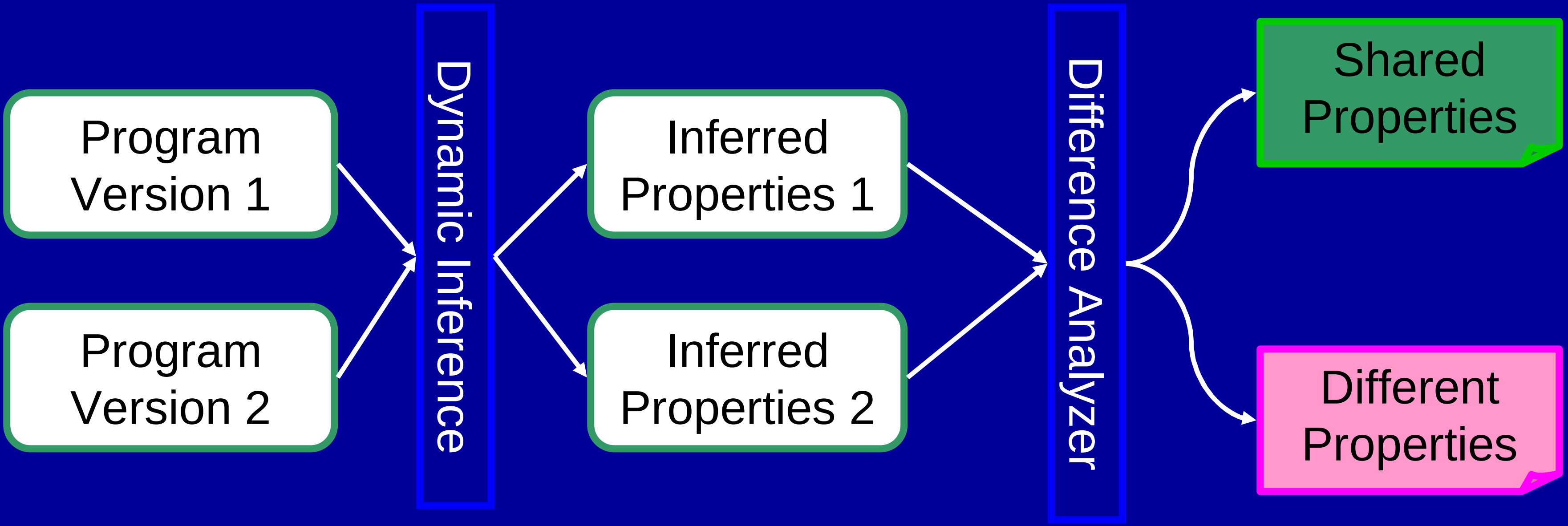
Jinlin Yang and David Evans. Dynamically inferring temporal properties. *PASTE '04*.

Implementation

- Terracotta
  - Scalable statistical inference
  - Context-aware analysis
  - Heuristics for prioritizing and presenting properties
- Complexity
  - Time: $O(nl)$, Space: $O(n^2)$
  - n: the number of distinct events
  - l: the length of the trace
- Available at:

  http://www.cs.virginia.edu/terracotta

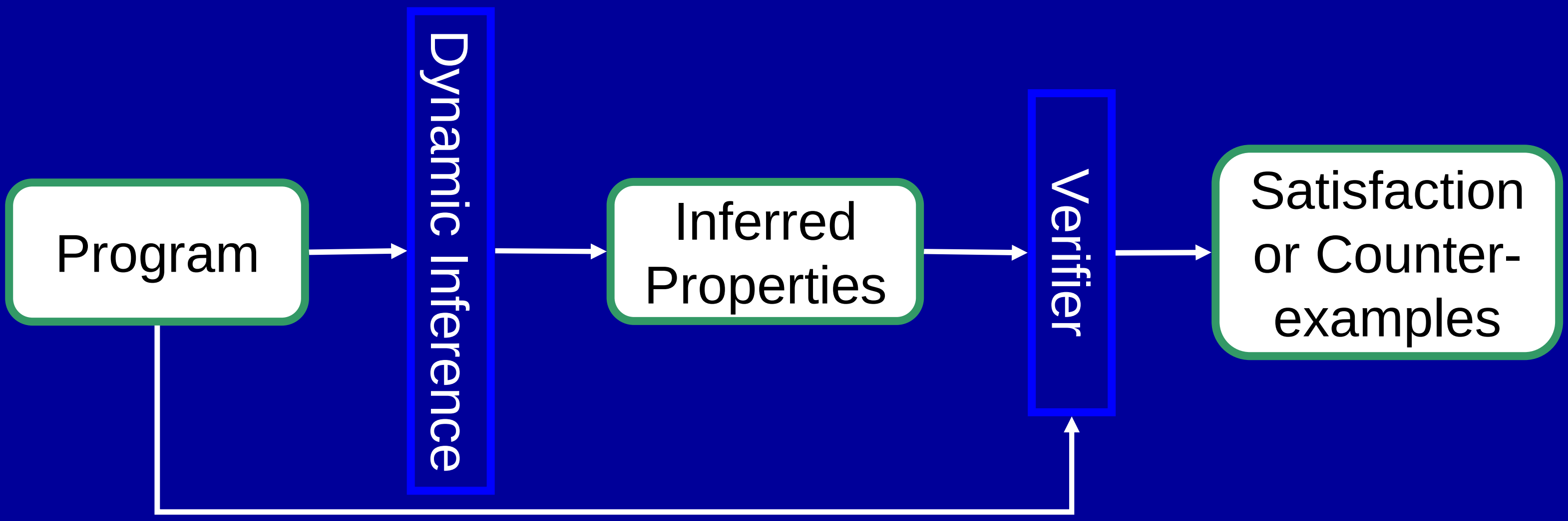
Use in Program Evolution



- Experiments on six versions of OpenSSL
  - Inferred an FSM conformant to the SSL specification
  - Revealed previously known bugs
  - Identified intended improvements

Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. *ISSRE '04*.

Use in Program Verification



- Inferred properties for the Daisy file system, then checked with Java PathFinder
  - Found one race condition
  - Revealed undocumented interesting differences of locking discipline across layers of the system

Jinlin Yang and David Evans. Automatically Discovering Temporal Properties for Program Verification. Technical Report, Department of Computer Science, University of Virginia, 2005.

Related Work

- Dynamic inference
  - **Daikon [Ernst, TSE01]**
  - **Mining specification [Ammons, POPL02]**
  - **FindLocks [Rose, SCP05]**
  - Encoding program executions [Reiss, ICSE01]
  - Recovering thread models [Cook, JSS04]
- Static inference
  - **Bugs as deviant behavior [Engler, SOSP01]**
  - **Extracting component interfaces [Whaley, ISSTA02]**
  - **Mining by examining exceptional path [Weimer, TACAS05]**
  - Houdini [Flanagan, FME01]
  - Synthesizing API interfaces [Alur, POPL05]
  - SALInfer [Hackett, MSR-TR-05]

Limitations of Previous Work

- Fail to find many important properties
  - When the traces are produced from buggy programs
  - Engler’s approach might miss properties of infrequent events
- Find too many uninteresting properties
  - Most inferred properties are useless
- Too slow
  - Trying to infer a complex FSM directly does not scale
- This talk is about a tool that overcomes these problems

Outline

- Introduction
- My approach
- **Preliminary experiment on Windows kernel APIs**
- Refinement of inference with new experimental results
- Contributions, future work, and conclusion

Experimental Setup

- 17 traces from developers
  - We had no control on producing the traces which were used for performance tuning or debugging
  - Converted into Terracotta's format
- Events
  - On average 500 distinct events (range from 40 to 1.3K)
  - Include non-kernel APIs (e.g. ntdll.dll, hal.dll)
- Trace length
  - Varies from 300K to 750K events
  - 5.85M events in total
- Terracotta finished analyzing in less than 14min

Results: Windows Kernel

- Some obviously interesting properties

ObpAllocateObjectNameBuffer->ObpFreeObjectNameBuffer
SeLockSubjectContext->SeUnlockSubjectContext
MmSecureVirtualMemory->MmUnsecureVirtualMemory
KefAcquireSpinLockAtDpcLevel->KefReleaseSpinLockFromDpcLevel
ExAcquireRundownProtectionCacheAwareEx->ExReleaseRundownProtectionCacheAwareEx
IoAcquireVpbSpinLock->IoReleaseVpbSpinLock
KeAcquireQueuedSpinLock->KeReleaseQueuedSpinLock

Lessons

- Missing interesting properties
  - KeAcquireInStackQueuedSpinLock-> KeReleaseInStackQueuedSpinLock
  - Original algorithm requires perfect traces
- Real world is never perfect :(
  - Imperfect programs
  - Trace collected by sampling
  - Object information unavailable
- Can we develop better inference to handle this?

Lessons (2)

- Too many noises in results
  - Interesting properties are buried in a group of uninteresting ones
- Can we develop heuristics to select interesting ones?

- The templates are small FSMs
  - FSMs in real world are usually bigger and more complex
- Can we develop techniques to construct bigger FSMs out of small ones?

Limitations of Previous Work Recap

- Fail to find many important properties
  - When the traces are produced from buggy programs
  - Engler's approach might miss properties of infrequent events
- Find too many uninteresting properties
  - Most inferred properties are useless
- Too slow
  - Trying to infer a complex FSM directly does not scale
- Terracotta scales very well to realistic traces

Outline

- Introduction
- My approach
- Preliminary experiment on Windows kernel APIs
- **Refinement of inference with new experimental results**
- Contributions, future work, and conclusion

Dealing with Reality

- How to infer interesting properties from imperfect traces?
- Example
  - *PSPSPSPSPSPSPSPSPSPPP*
  - The ***dominant*** behavior is *P* and *S* alternate
- How to define dominant?
  - *PS PS PS PS PS PS PS PS PS PPP*
  - 10 subtraces, 90% satisfy Alternating

Dealing with Reality (cont.)

- Definition of a subtrace
  - Intuitive: start with *P*, end with *S*
  - Formal: P^+S^+
- Decide if each subtrace satisfies Alternating
- Compute the Alternating percentage, P_{AL}
- Rank pairs of events based on P_{AL}
- Does not increase the complexity
  - Time: $O(nl)$, Space: $O(n^2)$
  - *n*: the number of distinct events
  - *l*: the length of the trace

Windows Kernel: Statistical Inference

P_{AL}	**Property (boldface ones are *not* in MSDN)**
0.9930	**ObpCreateHandle->ObpCloseHandle**
0.9880	**GreLockDisplay->GreUnlockDisplay**
0.9854	**RtlActivateActivationContextUnsafeFast->RtlDeactivateActivationContextUnsafeFast**
0.9821	KeAcquireInStackQueuedSpinLock->KeReleaseInStackQueuedSpinLock
0.9774	**SeCreateAccessState->SeDeleteAccessState**
0.9722	IoAllocateIrp->IoFreeIrp
0.9613	**CmpLockRegistry->CmpUnlockRegistry**
0.9589	**ObAssignSecurity->ObDeassignSecurity**
0.9565	**VirtualAllocEx->VirtualFreeEx**
0.9539	**ExCreateHandle->ExDestroyHandle**
0.9539	**ExpAllocateHandleTableEntry->ExpFreeHandleTableEntry**
0.9448	ExInitializeResourceLite->ExDeleteResourceLite

Selecting Properties: Using Call Graphs

- How to pick out interesting properties?

```
void A(){
  ...
  B();
  ...
}
```

Case 1

```
void x(){
  C();
  ...
  D();
}
```

Case 2

- Which one is more likely to be interesting?

Selecting Properties: Using Call Graphs

- How to pick out interesting properties?

```
void KeSetTimer(){
  KeSetTimerEx();
}
```

```
void x(){
  ExAcquireFastMutexUnsafe(&m);
  ...
  ExReleaseFastMutexUnsafe(&m);
}
```

- Which one is more likely to be interesting?

Selecting Properties: Using Call Graphs

- How to pick out interesting properties?

```
void A(){
  ...
  B();
  ...
}
    Case 1
```

```
void x(){
  C();
  ...
  D();
}
      Case 2
```

- Which one is more likely to be interesting?
  - Heuristics: C->D is often more interesting
  - Compute the static call graph for target programs
  - Keep A->B if B is not reachable from A

Selecting Properties: Edit Distance

- Heuristics: the more similar two events are, the more likely that the properties is interesting
- Relative edit distance between A and B
  - Partition A and B into words
  - A has w_A words, B has w_B, w common words
  - $$dist_{AB} = \frac{2w}{w_A + w_B}$$
- For example:
  - Ke Acquire In Stack Queued Spin Lock ->

    Ke Release In Stack Queued Spin Lock
  - Similarity = 85.7%

Windows Kernel: Applying Heuristics

- Approximation
  - P_{AL} threshold = 0.90
  - 7611 properties
- Call-graph and edit distance based reduction
  - Use the call-graph of ntoskrnl.exe, edit dist > 0.5
  - 142 properties. 53 times reduction!
  - Small enough for manual inspection
- 56 apparently interesting properties (40%)
  - Locking discipline
  - Resource allocation and deletion

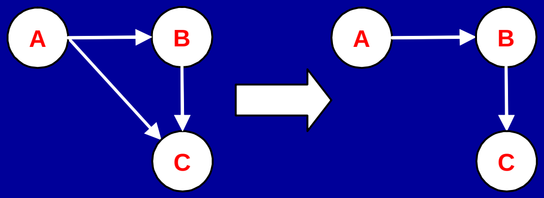
Windows Kernel: Usage of Properties

- Inferred useful properties that could be checked
  - Several types of kernel SpinLock
  - SLAM [Ball+ SPIN01] does not check two of them

- ESP [Das+ PLDI02] found many previously unknown bugs in Windows
  - E.g. Double-acquire of FastMutex in ntfs.sys
  - Found this one during my internship, confirmed and fixed by responsible developers
  - The group adopted the properties and found more bugs since I left

Constructing Larger FSMs

- How to construct big FSMs out of small ones?
- Chaining method
  - Explore the relationships among *Alternating* properties



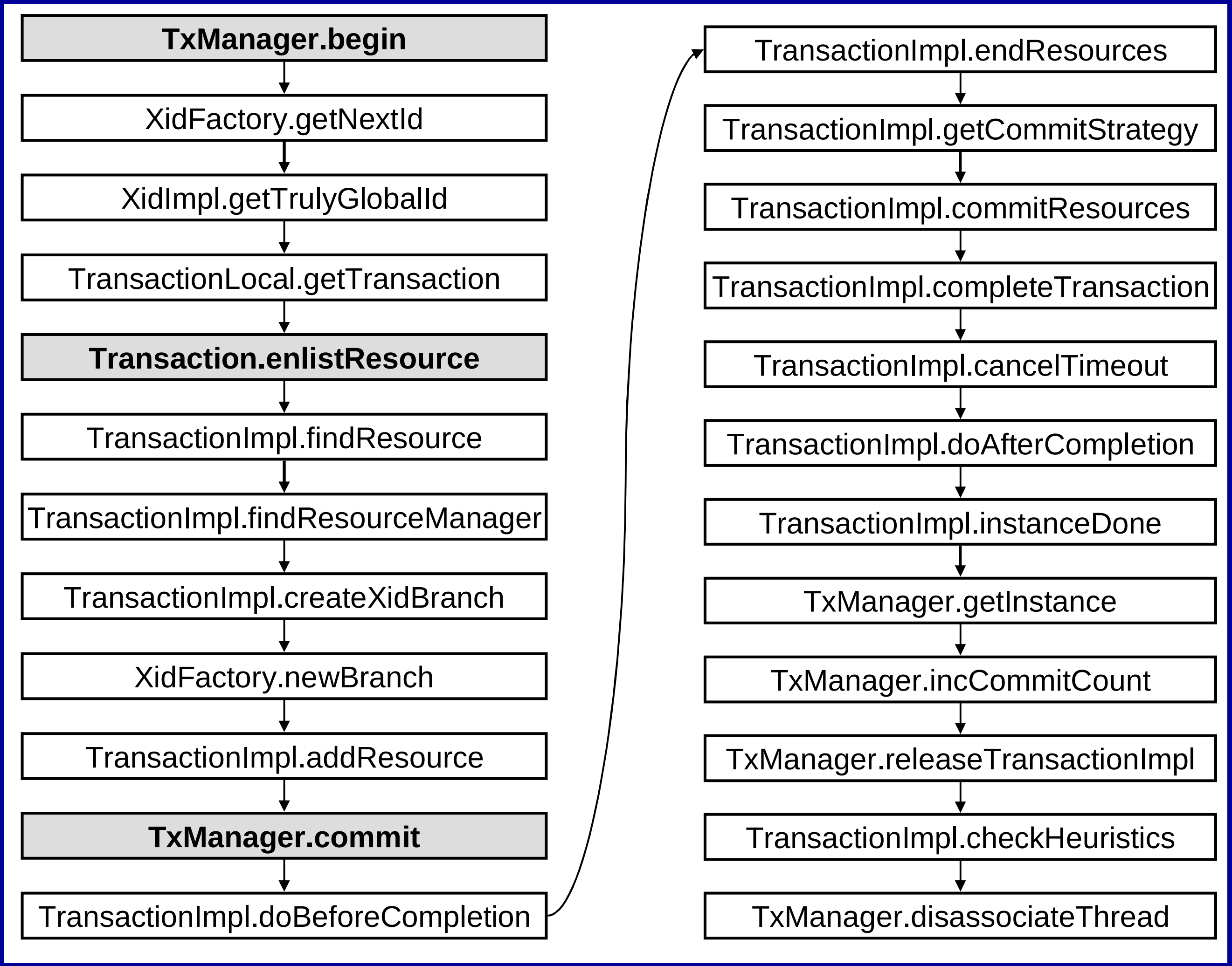
  - Potential reduction of the number of properties from $O(n^2)$ to $O(n)$
  - Efficiently producing more appealing results
  - The *Alternating* relation is not transitive

    For example: *ABACBC* -> *ABAB, BCBC, AACC*

Results of Chaining from JBoss

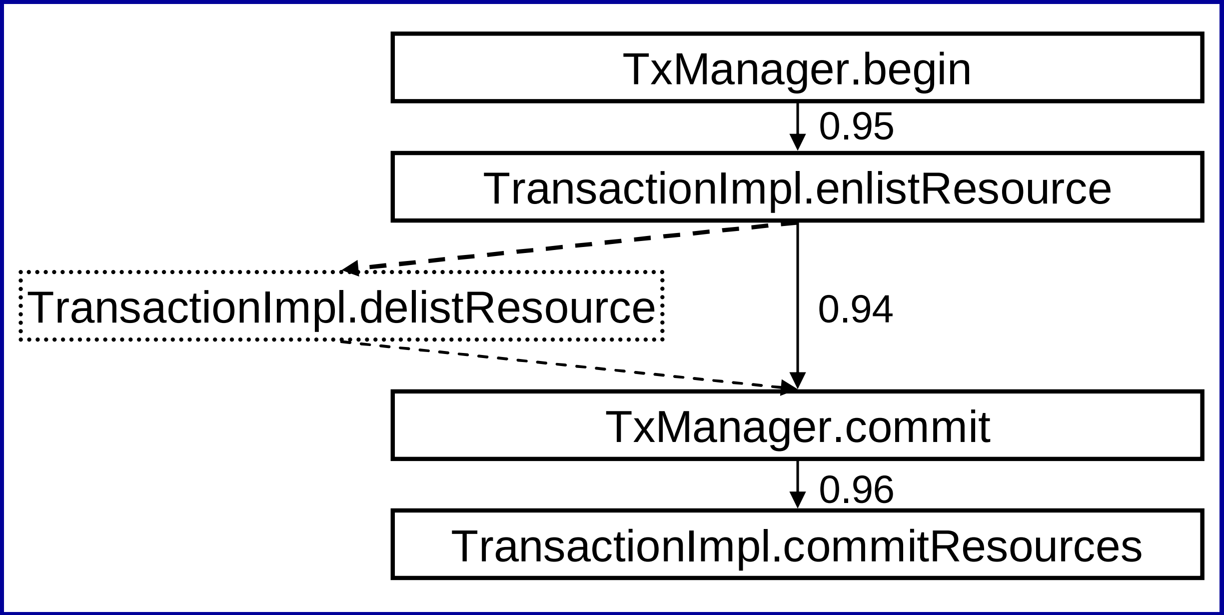
- Setup
  - A Java application server implementing J2EE
  - Instrumented the transaction manager module
  - Executed the JBoss regression test suite
  - 2.5 million events with 91 distinct events
  - Terracotta finished in 80 seconds
- Results
  - 490 properties when p_{AL}=0.90
  - 61 properties after chaining (17 chains)
  - 41 properties after call-graph reduction (16 chains)
  - Edit distance not very useful
  - The longest chain is consistent with the object interaction diagram in the Java Transaction API specification

JBoss: Chaining Properties



TxManager.begin
XidFactory.getNextId
XidImpl.getTrulyGlobalId
TransactionLocal.getTransaction
Transaction.enlistResource
TransactionImpl.findResource
TransactionImpl.findResourceManager
TransactionImpl.createXidBranch
XidFactory.newBranch
TransactionImpl.addResource
TxManager.commit
TransactionImpl.doBeforeCompletion
TransactionImpl.endResources
TransactionImpl.getCommitStrategy
TransactionImpl.commitResources
TransactionImpl.completeTransaction
TransactionImpl.cancelTimeout
TransactionImpl.doAfterCompletion
TransactionImpl.instanceDone
TxManager.getInstance
TxManager.incCommitCount
TxManager.releaseTransactionImpl
TransactionImpl.checkHeuristics
TxManager.disassociateThread

JBoss: Chaining Properties (2)



Summary of Experiments

- Approximation is essential in dealing with imperfect traces
  - 56 interesting rules of Windows kernel APIs
  - An 24-state FSM for JBoss
  - Rules undocumented by SLAM, SeLockSubjectContext -> SeUnlockSubjectContext

    http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/SDV-intro.doc
- Inference scales well to realistic traces
  - 5.85 million events, 500 distinct ones, 14 minutes
- Call-graph, edit distance, and chaining are very effective
  - Reduction: 53 times for Windows, 12 times for JBoss
  - An FSM for JBoss
- Check with defect detection tool is very promising
  - Many bugs found and fixed in Windows

Other Experiments

- Vulcan APIs
- Daisy file system [TR]
- Six versions of OpenSSL [ISSRE04]
- Submissions of programming assignments [ISSRE04]
- A simple producer-consumer implementation [PASTE04]

Outline

- Introduction
- My approach
- Preliminary experiment on Windows kernel APIs
- Refinement of inference with new experimental results
- **Contributions, future work, and conclusion**

Limitations of Previous Work Recap

- Fail to find many important properties
  - When the traces are produced from buggy programs
  - Engler's approach might miss properties of infrequent events
- Our approach can deal with imperfect traces and infer properties has low static frequency
- Find too many uninteresting properties
  - Most inferred properties are useless
- Our heuristics are very effective
  - A high percentage of the final properties are interesting
- Too slow
  - Trying to infer a complex FSM directly does not scale
- Our approach scales very well to realistic traces

Contributions Recap

- A statistical algorithm for inferring interesting temporal properties from imperfect traces
  - Windows kernel: 56 interesting properties
  - JBoss: an FSM consistent with the J2EE specification
- Two heuristics for eliminating uninteresting properties
  - Windows kernel: 53 times reduction, 40% are interesting
- Chaining method for constructing large FSMs
  - JBoss: an FSM with 24 states
- Combine automatic inference and verification together
  - ESP found many bugs in Windows using inferred properties
- Demonstration of effectiveness in realistic systems

Future Work

- More interesting and expressive property templates
  - Temporal property templates involving variables
  - E.g. between the start and end of the dispatch routine, deviceExtension.stopped should always be false
- Other ways to build large FSMs
  - Chains of mixed templates
- New ways to combine dynamic and static analysis
  - E.g. use static call graph to select interesting properties
  - Use dynamic analysis to make static analysis more scalable
  - Use static analysis to help testing, inference etc.

Conclusion

- Constructing interesting properties by hand is difficult

- Automatic inference from execution traces is effective
  - A statistical approach is essential for dealing with imperfect traces
  - Heuristics for identifying properties are important for practical use

- This approach has two practical uses
  - Understanding legacy code by inferring large FSMs from traces
  - Finding many application specific defects

Q & A

- For more information
  jinlin@cs.virginia.edu
  http://www.cs.virginia.edu/terracotta
- Great collaborators
  - UVa
    David Evans, Ed Mitchell
  - Microsoft
    Stephen Adams,
    Deepali Bhardwaj,
    Thirumalesh Bhat,
    Manuvir Das,
    Damian Hasse,
    Marne Staples, Rick Vicik,
    Jason Yang, Zhe Yang