272: Software Engineering
Fall 2018

Instructor: Tevfik Bultan

Lecture 14: Automated Interface Extraction

Software Interfaces

- Here are some basic questions about software interfaces
  - How to specify software interfaces?
  - How to check conformance to software interfaces?
  - How to extract software interfaces from existing software?
  - How to compose software interfaces?
- Today we will talk about some research that addresses these questions

Software Interfaces

- In this lecture we will talk about interface extraction for software components

- Interface of a software component should answer the following question
  - What is the correct way to interact with this component?
  - Equivalently, what are the constraints imposed on other components that wish to interact with this component?

- Interface descriptions in common programming languages are not very informative
  - Typically, an interface of a component would be a set of procedures with their names and with the argument and return types

Software Interfaces

- Let's think about an object oriented programming language
  - You interact with an object by sending it a message (which means calling a method of that object)
  - What do you need to know to call a method?
    - The name of the method and the types of its arguments
- What are the constraints on interacting with an object
  - You need a reference to the object
  - You have to have access (public, protected, private) to the method that you are calling
- One may want to express other kinds of constraints on software interfaces
  - It is common to have constraints on the order a component's methods can be called
    - For example: a call to the consume method is allowed only after a call to the produce method
  - How can we specify software interfaces that can express such constraints?

Software Interfaces

- Note that object oriented programming languages enforce one simple constraint about the order of method executions:
  - The constructor of the object must be executed before any other method can be executed
  - This rule is very static: it is true for every object of every class in every execution

- We want to express restrictions on the order of the method executions
  - We want a flexible and general way of specifying such constraints

Software Interfaces

- First, I will talk about the following paper
  - ``Automatic Extraction of Object-Oriented Component Interfaces," J. Whaley, M. C. Martin and M. S. Lam Proceedings of the International Symposium on Software Testing and Analysis, July 2002.
  - The following slides are based on the above paper and the slides from Whaley's webpage

Automatic Interface Extraction

- The basic idea is to extract the interface from the software automatically
  - Interface is not written as a separate specification
- There is no possibility of inconsistency between the interface specification and the code since the interface specification is extracted from the code
- The extracted interface can be used for dynamic or static analysis of the software
- It can be helpful as a reverse engineering tool

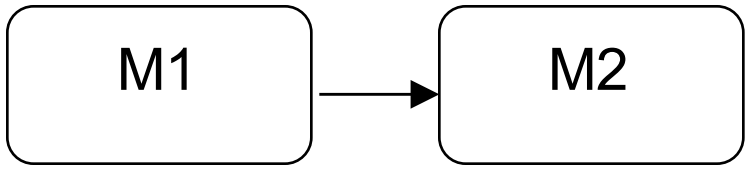
What are Software Interfaces

- In the scope of the work by Whaley et al. interfaces are constraints on the orderings of method calls

- For example:
  - method m1 can be called only after a call to method m2
  - both methods m1 and m2 have to be called before method m3 is called

How to Specify the Orderings

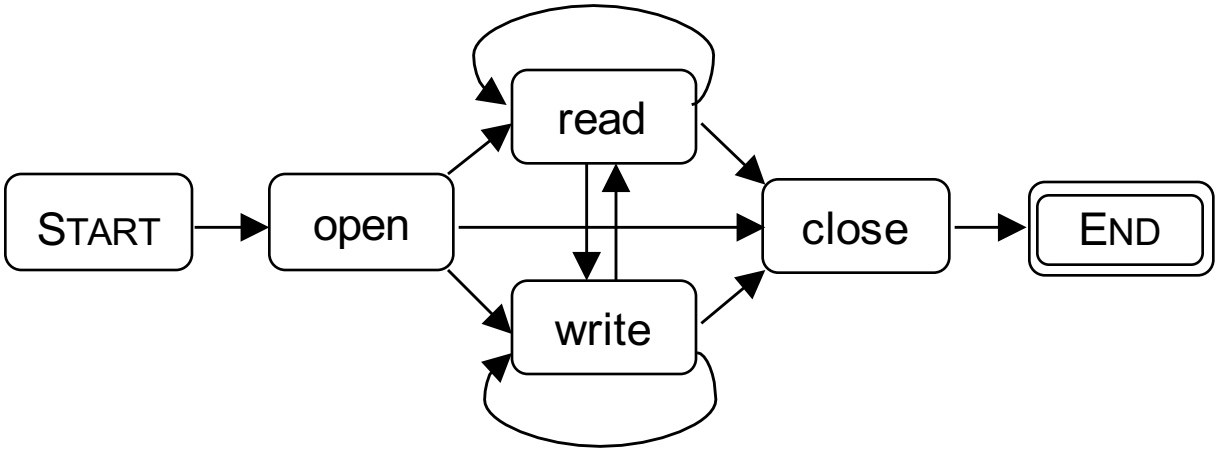
- Use a Finite State Machine (FSM) to express ordering constraints
- States correspond to methods
- Transitions imply the ordering constraints



Method M2 can be called after method M1 is called

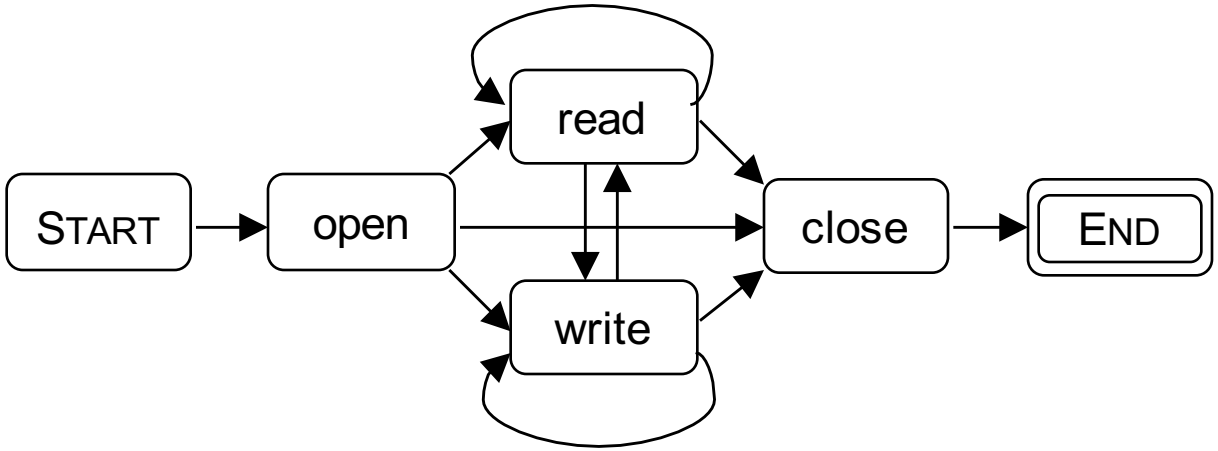
Example: File

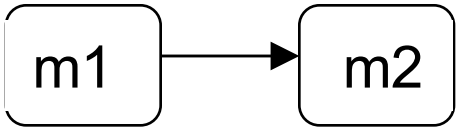
- There are two special states Start and End indicating the start and end of execution



A Simple OO Component Model

- Each object follows an FSM model.
- One state per method, plus START & END states.
- Method call causes a transition to a new state.





m1(); m2() is legal,
new state is m2

The Interface Model

- Note that this is a very simple model
- It only remembers what the last called method is
- There is no differentiation between different invocations of the same method
- This simple model reduces the number of possible states
- Obviously all the orderings cannot be expressed this way

Adding more precision

- With above model we cannot express constraints such as
  - Method m1 has to be called twice before method m2 can be called
- We can add more precision by remembering the last k method calls
  - If we have n methods this will create n^k states in the FSM
- Whaley et al. suggest other ways of improving the precision without getting this exponential blow up

The Interface Model

- If the only state information is the name of the last method called then what are the situations that this information is not precise enough?

- Problem 1: Assume that there are two independent sequences of methods that can be interleaved arbitrarily
  - once we call a method from one of the sequences we will lose the information about the other sequence

- Problem 2: Assume that there is a method which can be followed by all the other methods
  - then once we get to that method any following behavior is possible independent of the previous calls

Problem 1

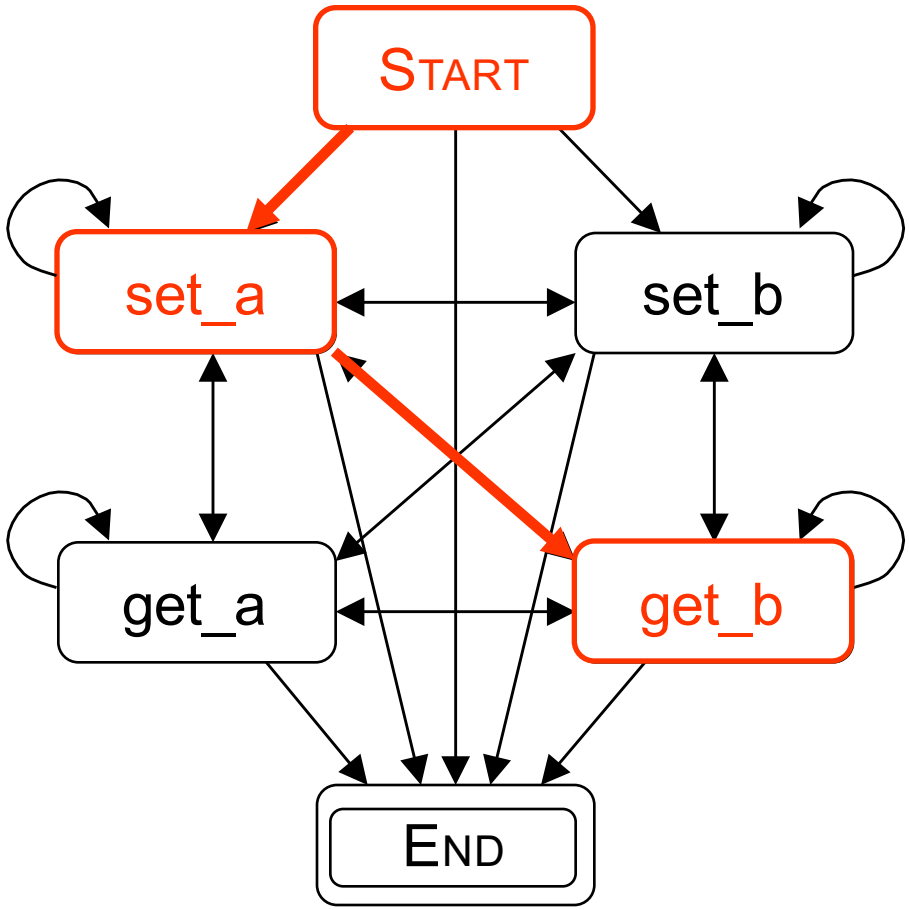
- Consider the following scenario
  - An object has two fields, a and b
  - There are four methods set_a(), get_a(), set_b(), get_b()
  - Each field must be set before being read

- We would like to have an interface specification that specifies the above constraints
  - Can we build an FSM that corresponds to these constraints?

Problem 1

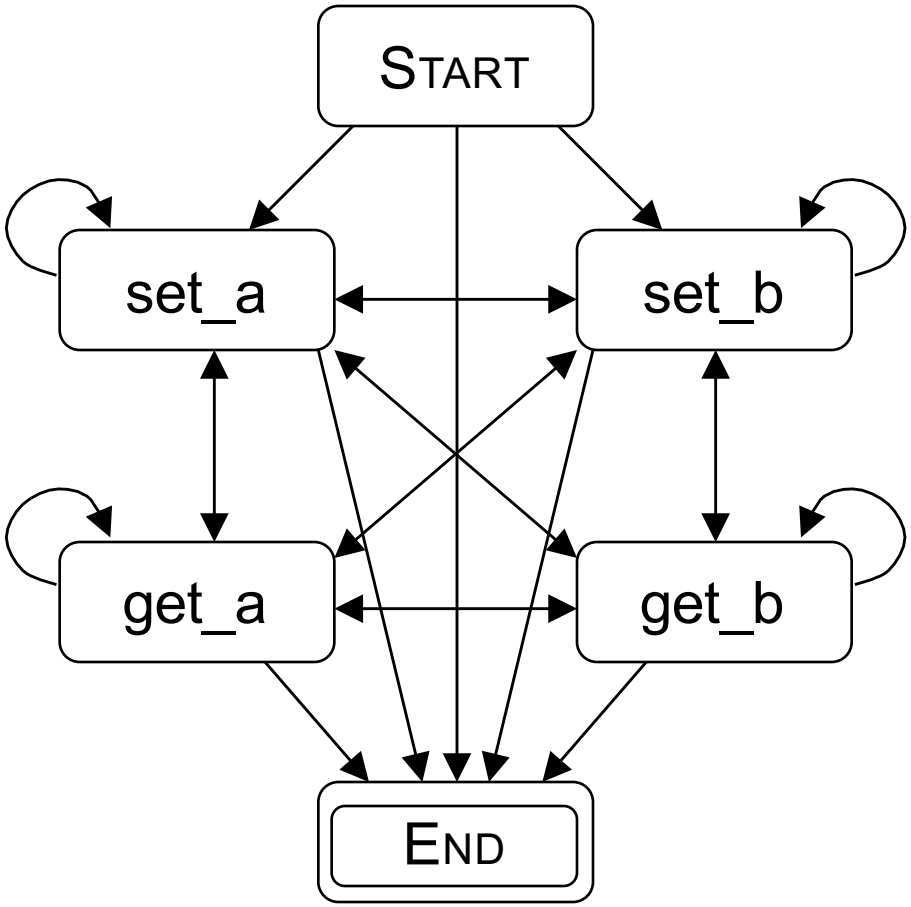
- These kind of constraints create a problem because once we call the set_a method it is possible to go to any other method
  - FSM does not remember the history of the method calls
  - FSM only keeps track of the last method call
- Solution: Use one FSM for each field and take their product

FSM below allows the following sequence: start; set_a(); get_b();

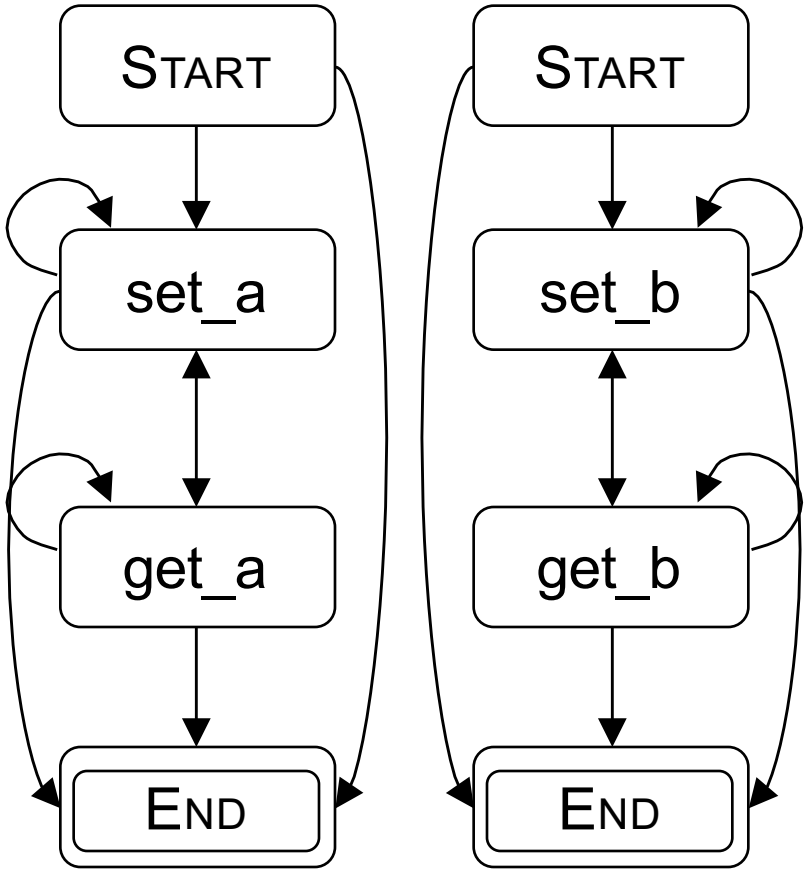


Splitting by fields

- Separate the constraints about different fields into different, independent constraints:
  - Use multiple FSMs executing concurrently (or use a product FSM)

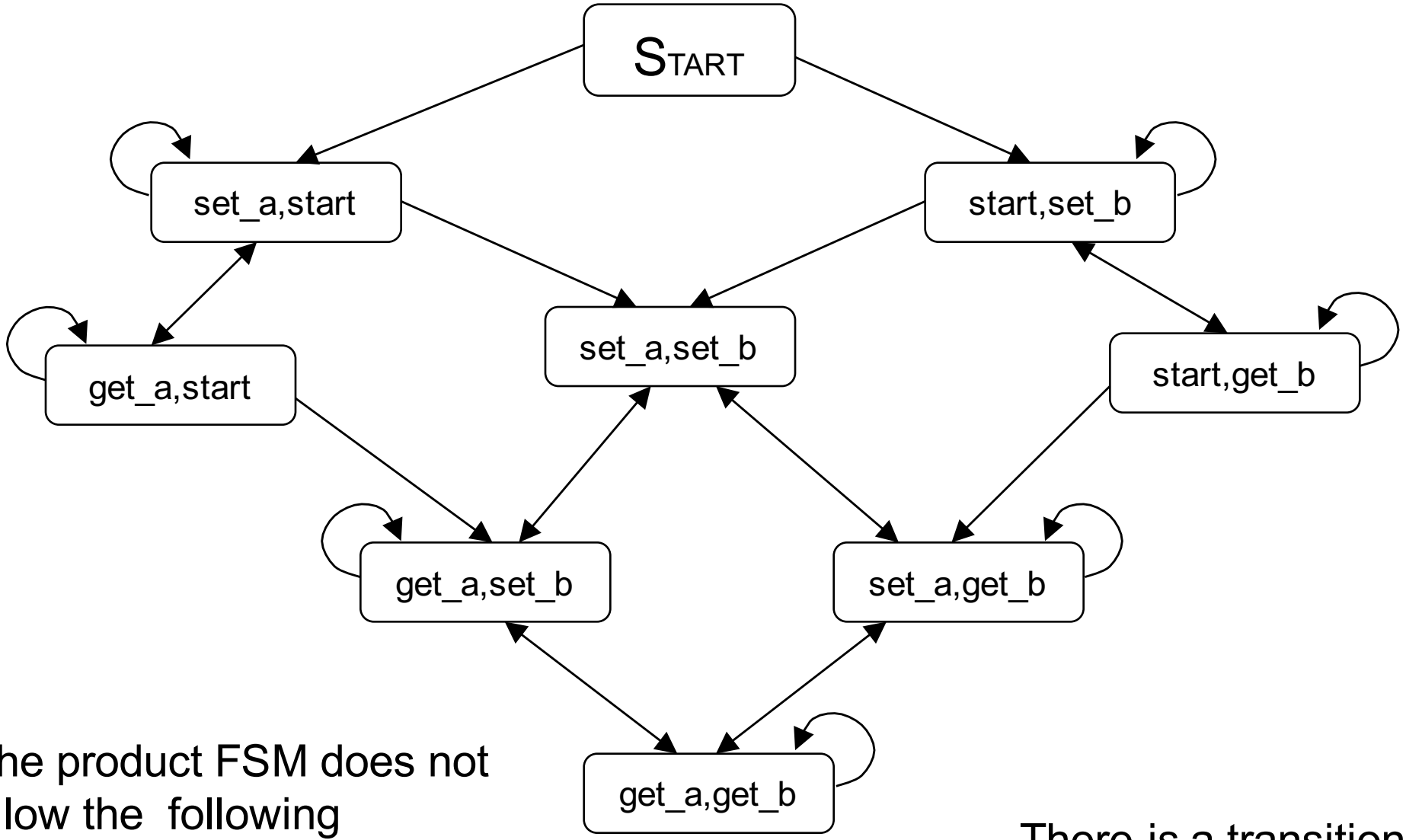


Imprecise



Adds more precision

Product FSM



The product FSM does not allow the following sequence:
start; set_a(); get_b();

There is a transition from each state to the END state

Product FSM

- Product FSM has more number of states than the FSM which just remembers the last call

- Assume that there are n_1 methods for field 1 and n_2 methods for field 2
  - simple FSM $n_1 + n_2$ states
  - product FSM $n_1 \times n_2$ states

- Note that the number states in the product FSM will be exponential in the number of fields

Problem 2

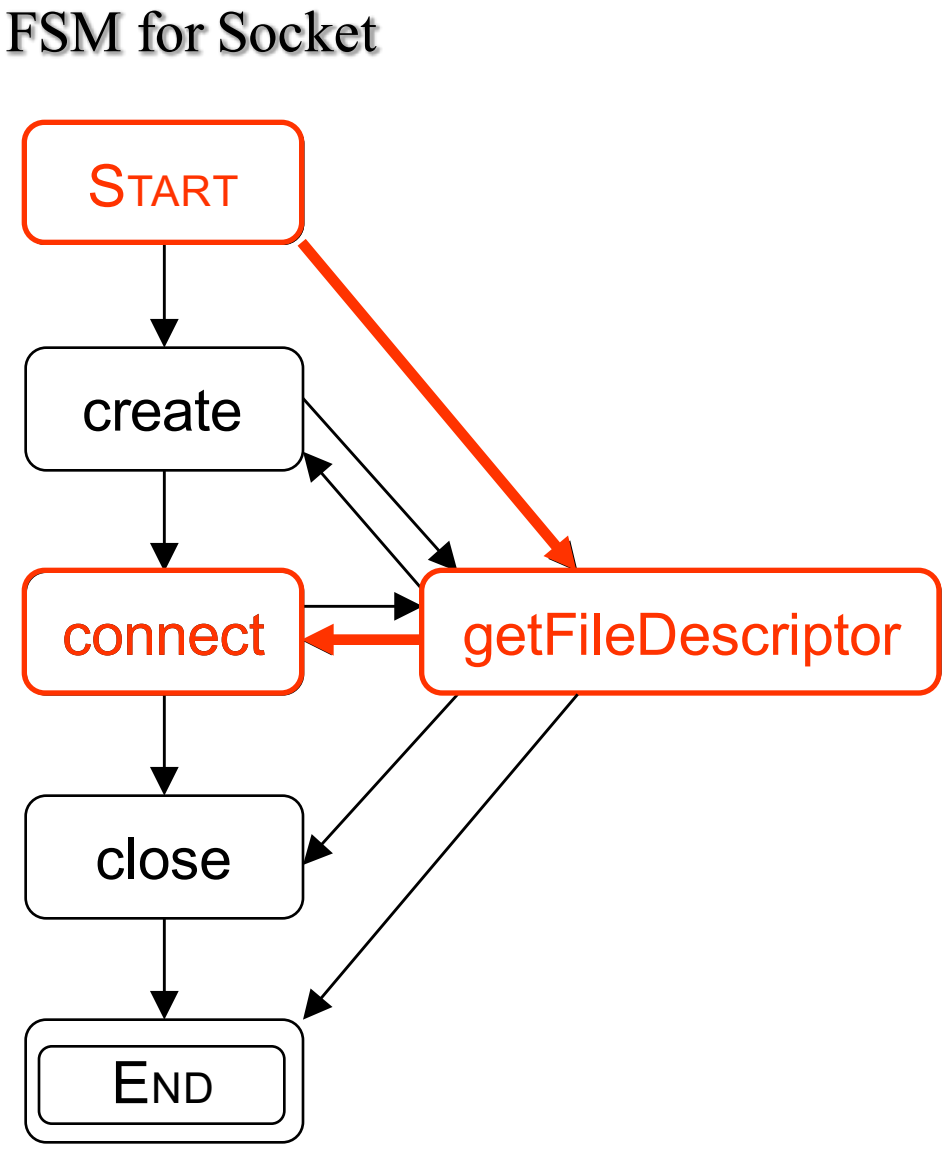
- It is common to have methods which are used to query the state of an object
- These methods do not change the state of the object
- After such state-preserving methods (aka query methods) all other methods can be called
  - Calling a state preserving method does not change the state of the object
  - If a method can be called before a call to a state preserving method, then it can be called after the call to the state preserving method
  - Since only information we keep in the FSM is the last method call, if there exists an object state where a method can be called, then that method can also be called after a call to a state-preserving method

Problem 2

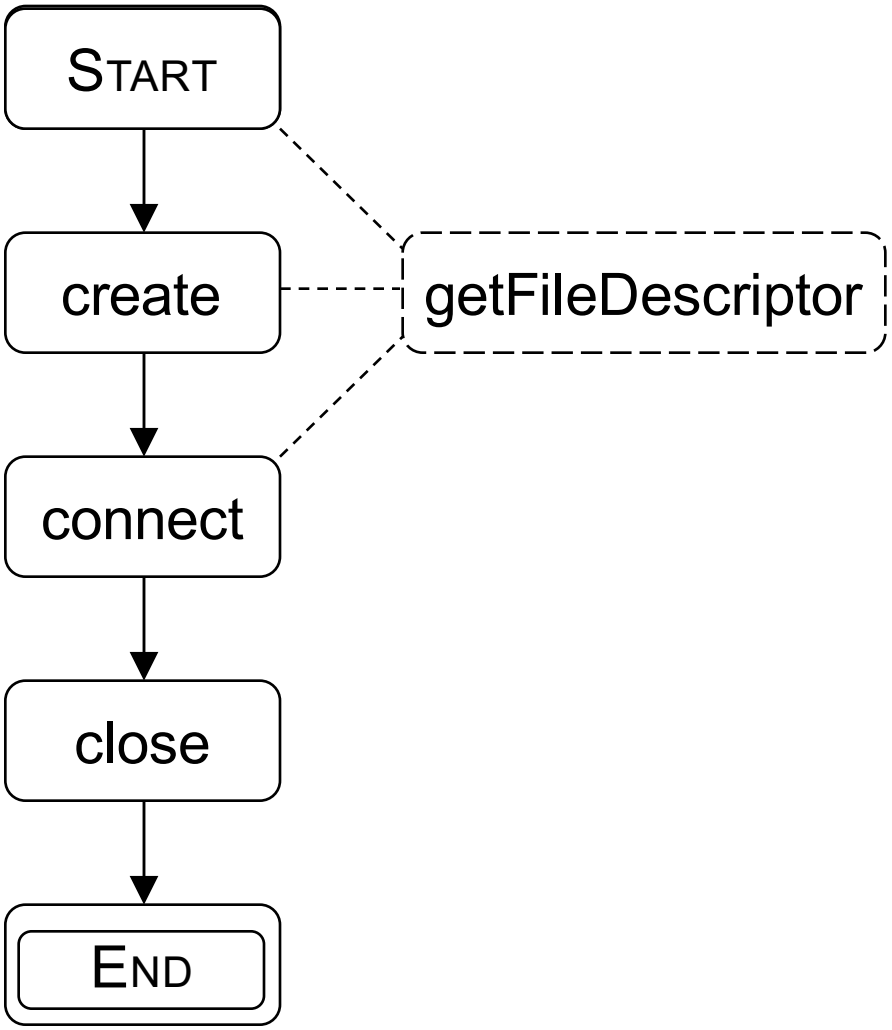
- getFileDescriptor is state-preserving
- Once getFileDescriptor is called then any behavior becomes possible
- The FSM for Socket allows the sequence:

start; getFileDescriptor(); connect();

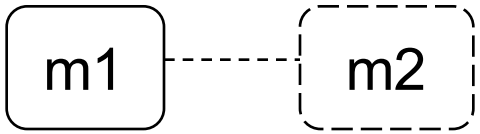
- Solution
  - distinguish between state-modifying and state-preserving methods
  - Calls to state-preserving methods do not change the state of the FSM



State-preserving methods



Calls to state-preserving methods do not change the state of the FSM



m1 is state-modifying
m2 is state-preserving
m1(); m2() is legal,
new state is m1

Summary of Model

- Product of FSMs
  - Per-thread, per-instance

- One submodel per field
  - Use static analysis to find the methods that either read the value of the field or modify the value of the field.
    - Identifies the methods that belong to a submodel
      - The methods that read and write to a field will be in the FSM for that field
    - Separates state-modifying and state-preserving methods.

- One submodel per Java interface
  - Implementation not required

Extraction Techniques

Static	Dynamic
For all possible program executions	For one particular program execution
Conservative	Exact (for that execution)
Analyze implementation	Analyze component usage
Detect illegal transitions	Detect legal transitions
Superset of ideal model (upper bound)	Subset of ideal model (lower bound)

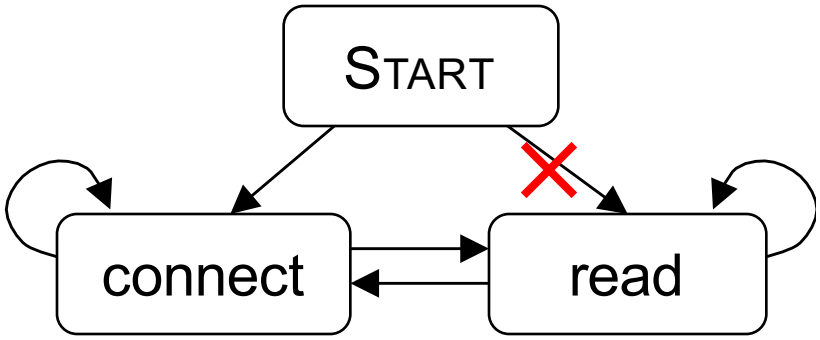
Static Model Extraction

- Static model extraction relies on defensive programming style
- Programmers generally put checks in the code that will throw exceptions in case the methods are not used in the correct order
- Such checks implicitly encode the software interface
- The static extraction algorithm infers the method orderings from these checks that come from defensive programming

Static Model Extractor

- Defensive programming
  - Implementation throws exceptions (user or system defined) on illegal input.

```
public void connect() {
   connection = new Socket();
}
public void read() {
   if (connection == null)
     throw new IOException();
}
```



Extracting Interface Statically

- The static algorithm has two main steps
  1. For each method m identify those fields and predicates that guard whether exceptions can be thrown
  2. Find the methods m’ that set those fields to values that can cause the exception
     - This means that immediate transitions from m’ to m are illegal
     - Complement of the illegal transitions forms the model of transitions accepted by the static analysis

Detecting Illegal Transitions

- Only support simple predicates
  - Comparisons with constants, null pointer checks

- The goal is to find method pairs <source, target> such that:
  - Source method executes:
    - *field* = *const* ;
  - Target method executes:
    - if (*field* == *const)*
      throw exception;

Algorithm

- How to find the target method: Control dependence
  - Find the following predicates: A predicate such that throwing an exception is control dependent on that predicate
    - This can be done by computing the control dependence information for each method
  - For each exception check if the predicate guarding its execution (i.e., the predicate that it is control dependent on) is
    - a single comparison between a field of the current object and a constant value
    - the field is not written in the current method before it is tested
  - Such fields are marked as state variables

Algorithm

- The second step looks for methods which assign constant values to state variables

- How to find the source method: Constant propagation
  - Does a method set a field to a constant value always at the exit?

- If we find such a method and see that
  - that constant value satisfies the predicate that guards an exception in an other method
    - then this means that we found an illegal transition

Sidenote: Control Dependence

- A statement S in the program is control dependent on a predicate P (an expression that evaluates to true or false) if the evaluation of that predicate at runtime may decide if S will be executed or not

- For example, in the following program segment

```
if (x > y) max:=x; else max:=y;
```

  the statements `max:=x;` and `max:=y;` are control dependent on the predicate `(x > y)`

- A common compiler analysis technique is to construct a control dependence graph
  - In a control dependence graph there is an edge from a node n_1 to another node n_2 if n_2 is control dependent on n_1

Sidenote: Constant Propagation

- Constant propagation is a well-known static analysis technique
- Constant propagation statically determines the expressions in the program which always evaluate to a constant value
- Example

```
y:=0; if (x > y) then x:=5; else x:=5+y; z := x*x;
```

  The assigned value to z is the constant 25 and we can determine this statically (at compile time)
- Constant propagation is used in compilers to optimize the generated code.
  - Constant folding: If an expression is known to have a constant value, it can be replaced with the constant value at compile time preventing the computation of the expression at runtime.

Static Extraction

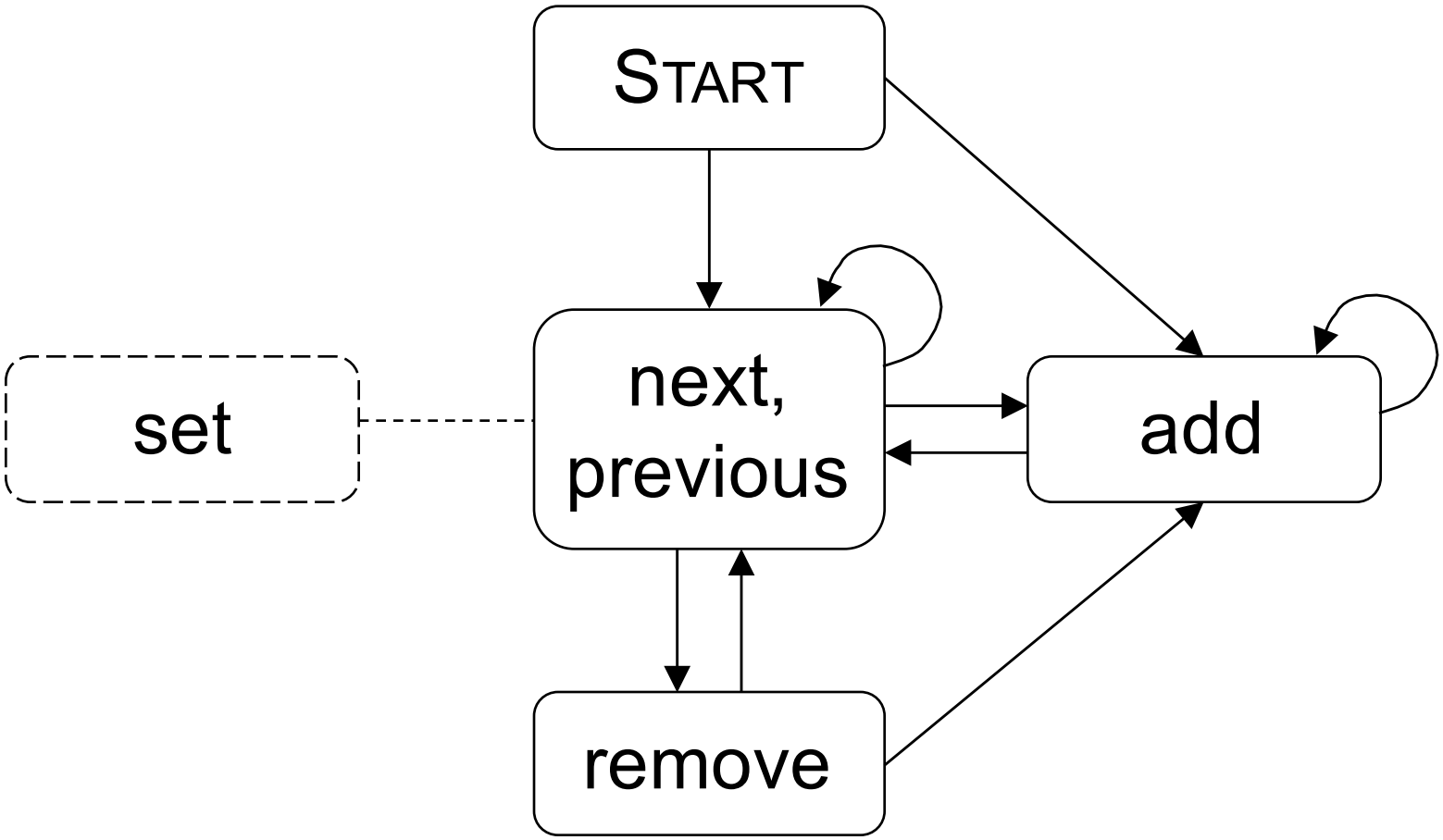
- Static analysis of the java.util.AbstractList.ListItr with lastRet field as the state variable

- The analysis identifies the following transitions illegal:
  - start →set
  - start→remove
  - remove→set, add→set
  - remove→remove
  - add→remove

- The interface FSM contains all the remaining transitions

Automatic documentation

- Interface generated for java.util.AbstractList.ListItr



Dynamic Interface Extractor

- Goal: find the legal transitions that occur during an execution of the program

- Java bytecode instrumentation
  - insert code to the method entry and exits to track the last-call information

- For each thread, each instance of a class:
  - Track last state-modifying method for each submodel.

Dynamic Interface Checker

- Dynamic Interface Checker uses the same mechanism as the dynamic interface extractor

  - When there is a transition which is not in the model
    - instead of adding it to the model
    - it throws an exception

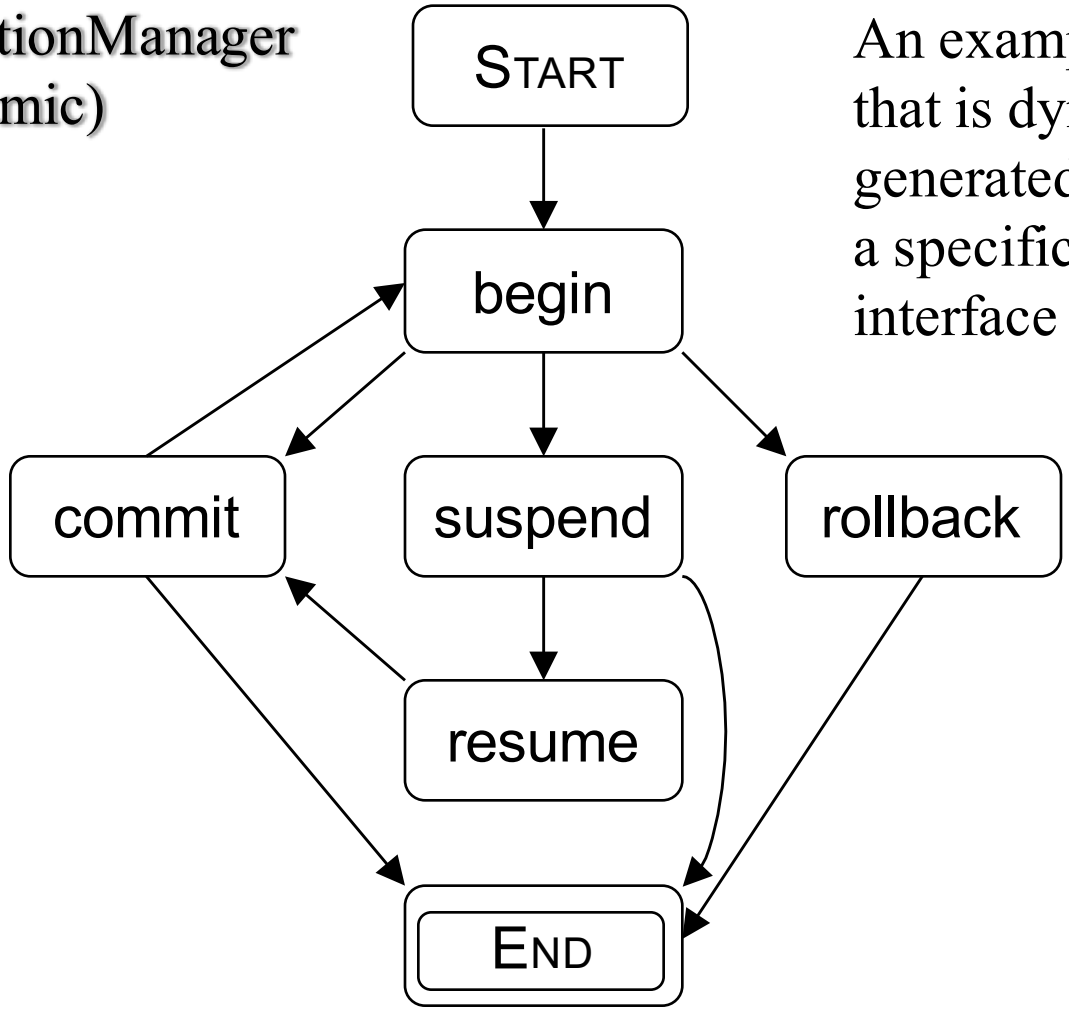
Experiences

- Whaley et al. applied these techniques to several applications

Program	Description	Lines of code
Java.net 1.3.1	Networking library	12,000
Java libraries 1.3.1	General purpose library	300,000
J2EE 1.2.1	Business platform	900,000
joeq	Java virtual machine	65,000

Automatic documentation

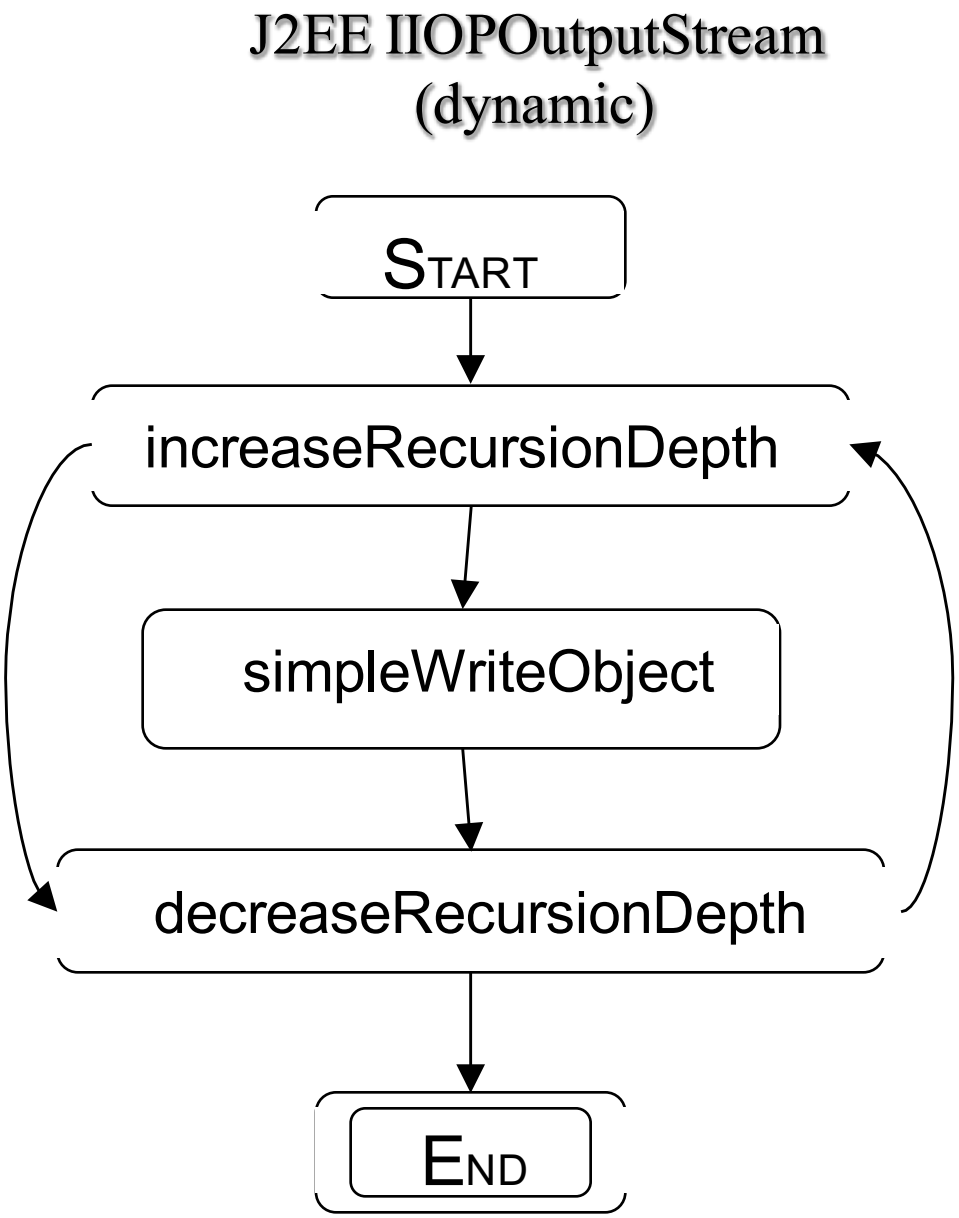
J2EE TransactionManager
(dynamic)



An example FSM model that is dynamically generated and provides a specification of the interface

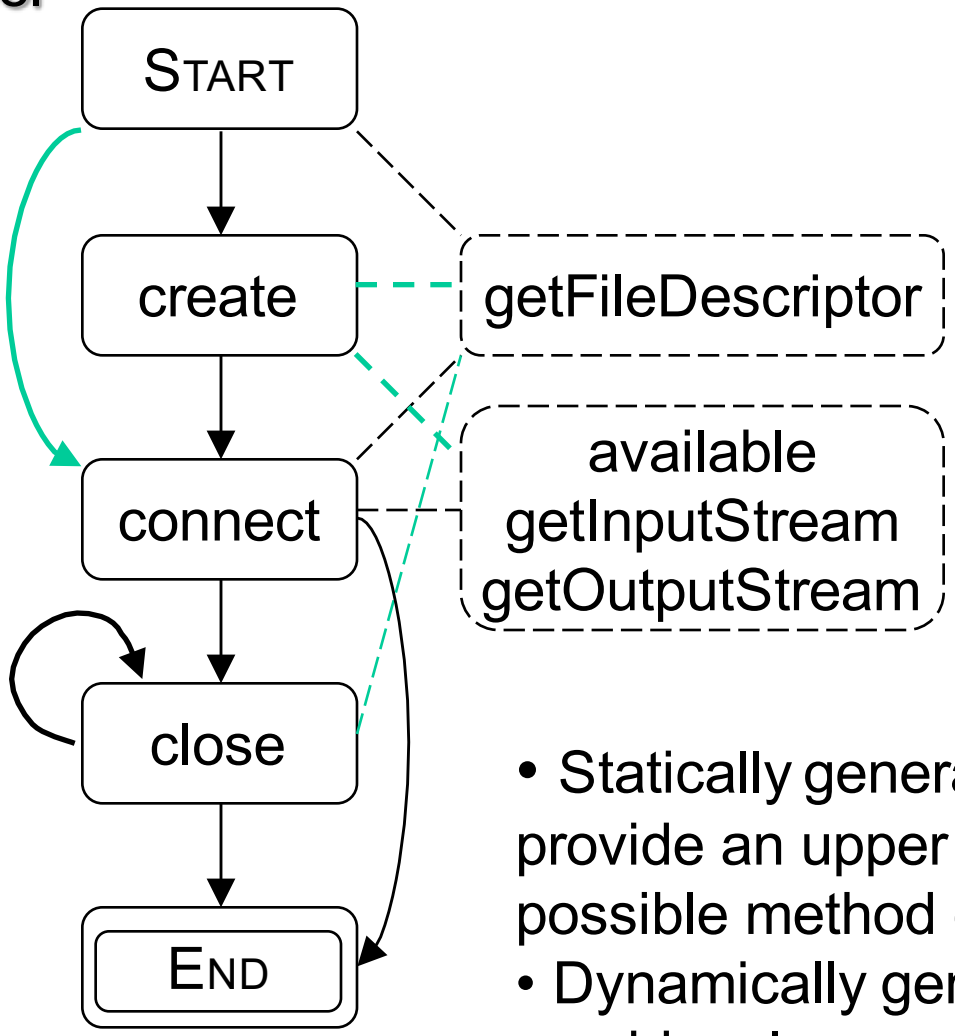
Test coverage

- Dynamically extracted interfaces can be used as a test coverage criteria
- The transitions that are not present in the interface imply that those method call sequences were not generated by the test cases
- For example, the fact that there are no self-edges in the FSM on the right implies that only a max recursion depth of 1 was tested



Upper/lower bound of model

SocketImpl model
(dynamic)
(+static)

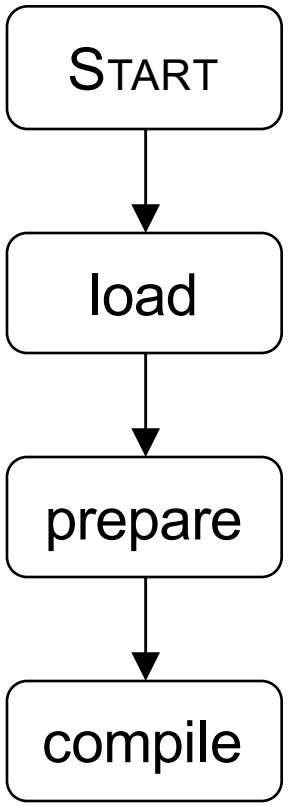


- Statically generated transitions provide an upper approximation of the possible method call sequences
- Dynamically generated transitions provide a lower approximation of the possible method call sequences

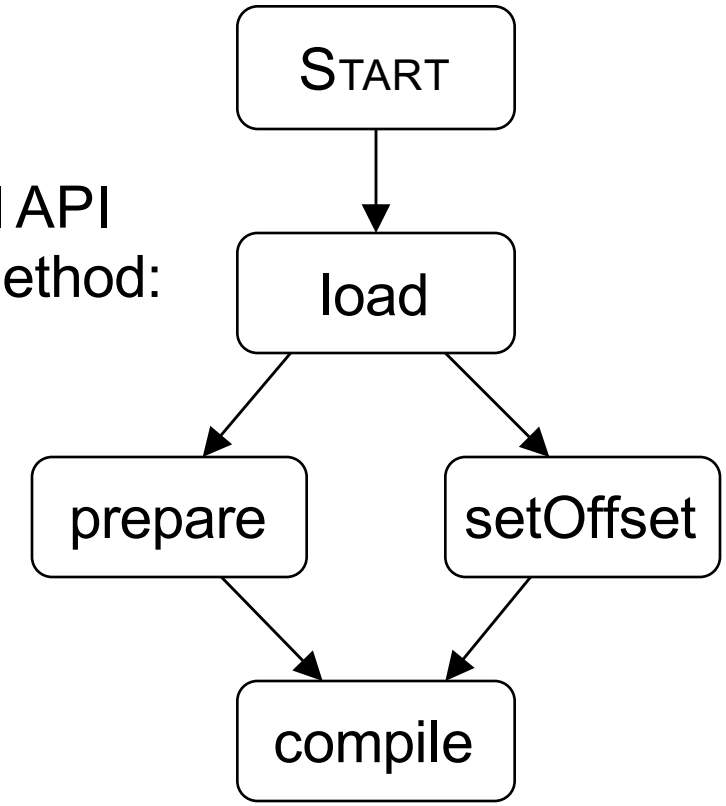
Finding API bugs

- Automated interface extraction can be used to detect bugs
- The interface extracted from the joeq virtual machine showed unexpected transitions

Expected API for jq_Method:



Actual API for jq_Method:



Summary: Automatic Interface Extraction

- Product of FSM
  - Model is simple, but useful
- Static and dynamic analysis techniques
  - Generate upper and lower bounds for the interfaces
- Useful for:
  - Documentation generation
  - Test coverage
  - Finding API bugs

Automated Interface Extraction, Continued

- There has been a lot of work on automated interface extraction. Here is another work on interface extraction
  - "Static Specification Mining Using Automata-Based Abstractions" Sharon Shoham, Eran Yahav, Stephen Fink, Marco Pistoia. International Symposium on Software Testing and Analysis (ISSTA 2007).

I will discuss this work in the rest of the lecture.

Representing interfaces with automata

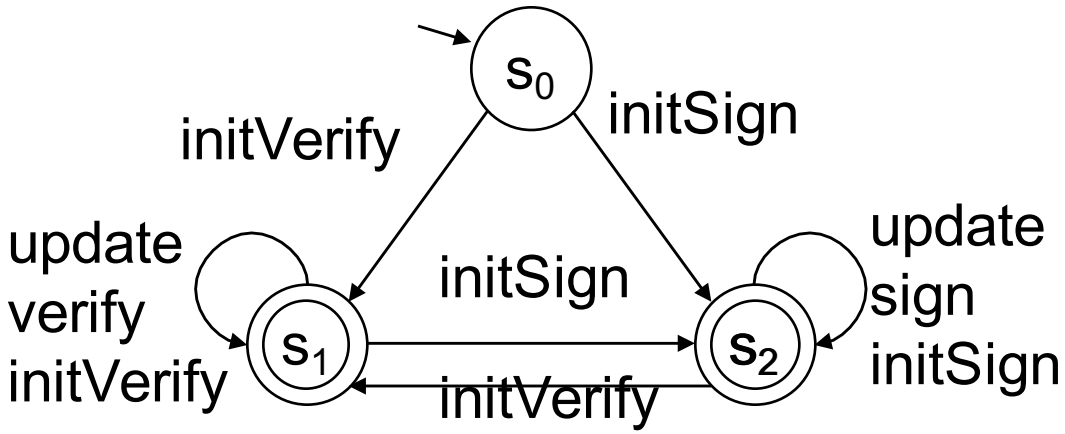
- In the earlier paper we discussed we saw a way to represent interfaces as state machines where the states represent the last called method and the transitions are not labeled

- Another way to represent interfaces is to use automata as an interface specification
  - The method names form the alphabet of the automata
  - The transitions are labeled with method names
  - An interface automaton accepts the allowable sequences of method calls

Representing interfaces with automata

- The automata representing the interface for the Java class Signature
- The method calls are represented by the transitions
- The paths from the initial state to accepting states identify the acceptable method call sequences

Signature class
interface



Component vs. Client-side extraction

- Work on interface extraction can be categorized as component-side or client-side extraction

- Component-side extraction: Analyze the component code and based on the error conditions in the component code (such as thrown exceptions) identify the illegal method call orderings
  - This was the approach used in the first paper we discussed

- Client-side extraction: Analyze a set of client-code that uses the component. Identify the ordering of method calls from the clients to the component and generate an interface that summarizes all the ways the clients interact with the component
  - This is the approach used in the second paper

Static Analysis for Interface Extraction

- The first paper we discussed used both static and dynamic analysis
- The second paper uses only static analysis to extract the interfaces
- There are many variants of static analysis (ones on the left are less precise variants):
  - intra-procedural vs. inter-procedural
  - flow-insensitive vs. flow-sensitive
  - context-insensitive vs. context-sensitive
  - without alias analysis vs. with alias analysis

- In the client-side interface extraction work they use the more precise variants
  - Static analysis techniques over-approximate program behavior,
    - less precise analysis leads to more coarse approximation
  - Precise static analysis is more expensive
    - takes more time and memory

Techniques for Interface extraction

- Heap abstraction: Combine the behavior of objects that are allocated at the same program location
- Trace abstraction: Merge states in the automaton that have equivalent past/future behaviors
  - Defined based on the sequences of incoming or outgoing transitions
- Merge: Merge automata with similar behavior
  - Similarity can be defines using past/future abstractions
- Noise Reduction: Remove transitions that are only observed in a small number of clients
  - Could be erroneous use of the API