Software Component Protocol Inference

Tao Xie

General Examination Presentation

Dept. of Computer Science and Engineering

University of Washington

6 June 2003

Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions

Background

- Software component
  - “defined as a unit of composition with contractually specified interfaces and explicit context dependencies only.” [Szyperski98]
- Component interface
  - Services that the component provides to and requests from other components
- Component interface protocol/component protocol
  - Sequencing constraints on the interface (bi-directional)

Focus

- Components written in OO languages
- Unidirectional protocol

Example: `java.util.zip.zipOutputStream`

```
public class ZipOutputStream
         extends DeflaterOutputStream implements ZipConstants {
    public ZipOutputStream(OutputStream out);
    public static final int DEFLATED;
    public static final int STORED;
    public void close() throw IOException;
    public void closeEntry() throw IOException;
    public void finish () throws IOException;
    public void putNextEntry(ZipEntry e) throws IOException;
    public void setComment(String comment);
    public void setLevel(int level);
    public void setMethod(int method);
    public synchronized void write(byte[] b, int off, int len) throws
       IOException;
}
```

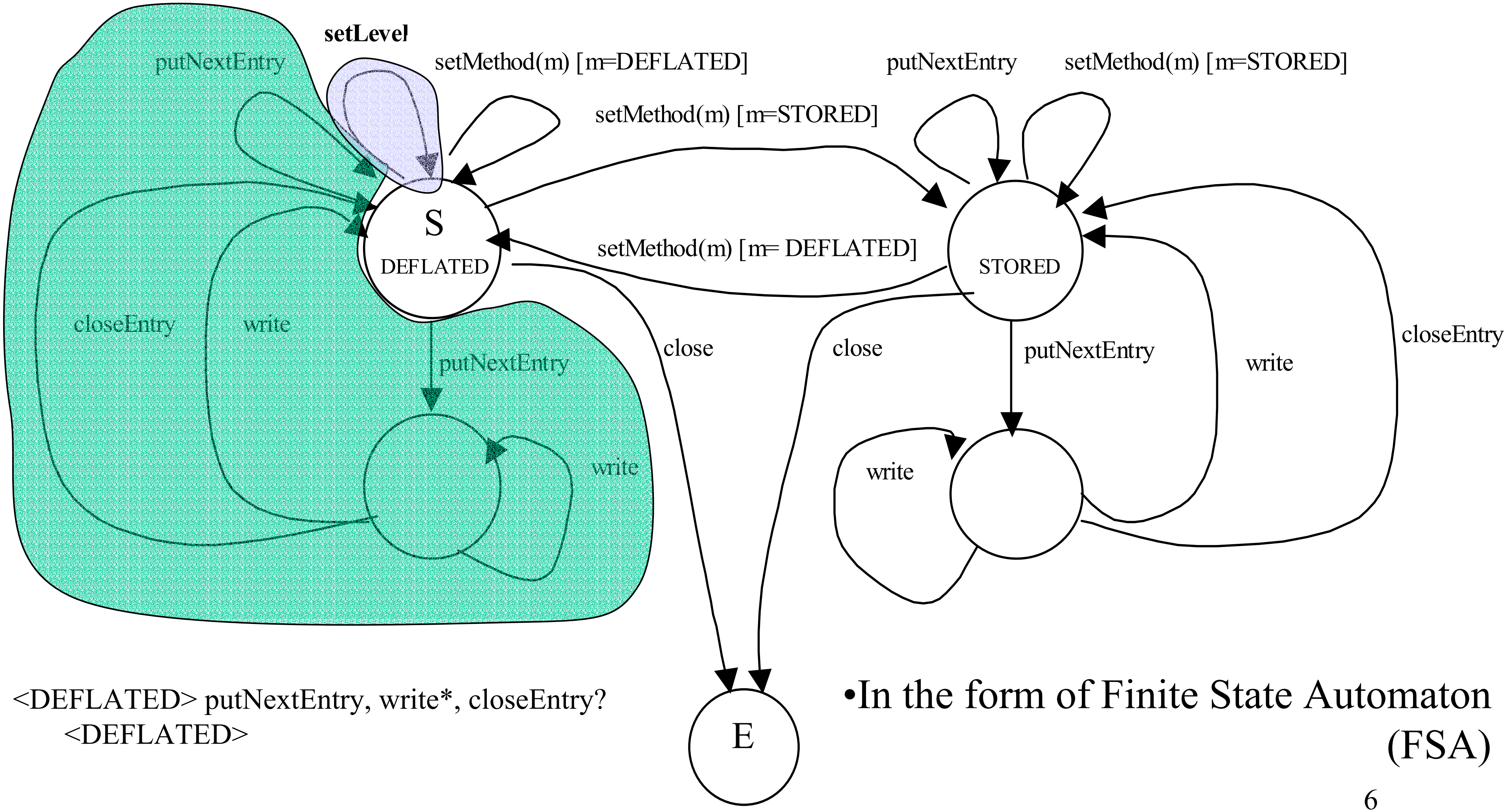
Informal Documentation

- from *Java in a Nutshell* [Flanagan97]

Once you have begun an entry with `putNextEntry()`,	you can write the contents of that entry with the `write()` methods.
When you reach the end of an entry,	you can begin a new one by calling `putNextEntry()` again, or you can close the current entry with `closeEntry()`, or you can close the stream itself with `close()`.
Before beginning an entry with `putNextEntry()`,	you can set the compression method and level with `setMethod()` and `setLevel()`.
The constants `DEFLATED` and `STORED` are the two legal values for `setMethod()`. If you use `STORED`, the entry is stored in the ZIP file without any compression.	
If you use `DEFLATED` [for `setMethod()`],	you can also specify the compression speed/strength tradeoff bypassing a number from 1 to 9 to `setLevel()`.

Formal Protocol Specification

- Translated from [Butkevich et al. 00]



<DEFLATED> putNextEntry, write*, closeEntry?
<DEFLATED>

- In the form of Finite State Automaton (FSA)

Why Component Protocol Inference?

- Protocols are useful for correct component usage
  - Documentation
  - Static verification
  - Runtime verification
- But few components have accompanying protocols

Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions

Protocol Inference

- Dynamic protocol inference
  - Inputs
    - Traces of method calls in the interface
- Static protocol inference
  - Inputs
    - Component code implementing the interface
    - Client code using the interface

Overview of Previous Work

Previous work	Target lang/sys	Analysis type	Result
Whaley et al. [WML02]	Java	Static and Dynamic	FSA
Reiss et al. [RR01]	Java, C++, and C	Dynamic	FSA
Ammons et al. [ABL02]	C	Dynamic	FSA
Cook et al. [CW98]	Software process	Dynamic	FSA
El-Ramly et al. [ESS02]	Interactive system	Dynamic	Frequently recurring usage patterns
Lie et al. [LCED01]	C protocol code	Static	FSA-like models to a model checker

Challenges

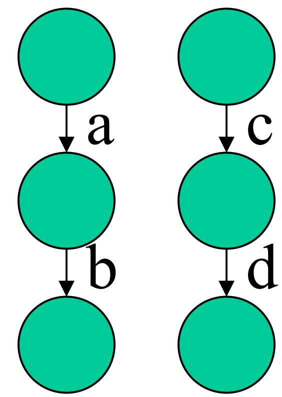
- Overgeneralization/over-restrictiveness
  - Overgeneralization: accept some illegal sequences
  - Over-restrictiveness: reject some legal sequences

•Separation/composition of constraints

–e.g. `DEFLATED` and `STORED` groups

–e.g. Concurrent FSAs

Interface:a,b,c,d,e



•Data-dependent transitions

–e.g. `setMethod(DEFLATED),setMethod(STORED)`

–e.g. `pop()` when `currentSize>0`

•Robustness to noise

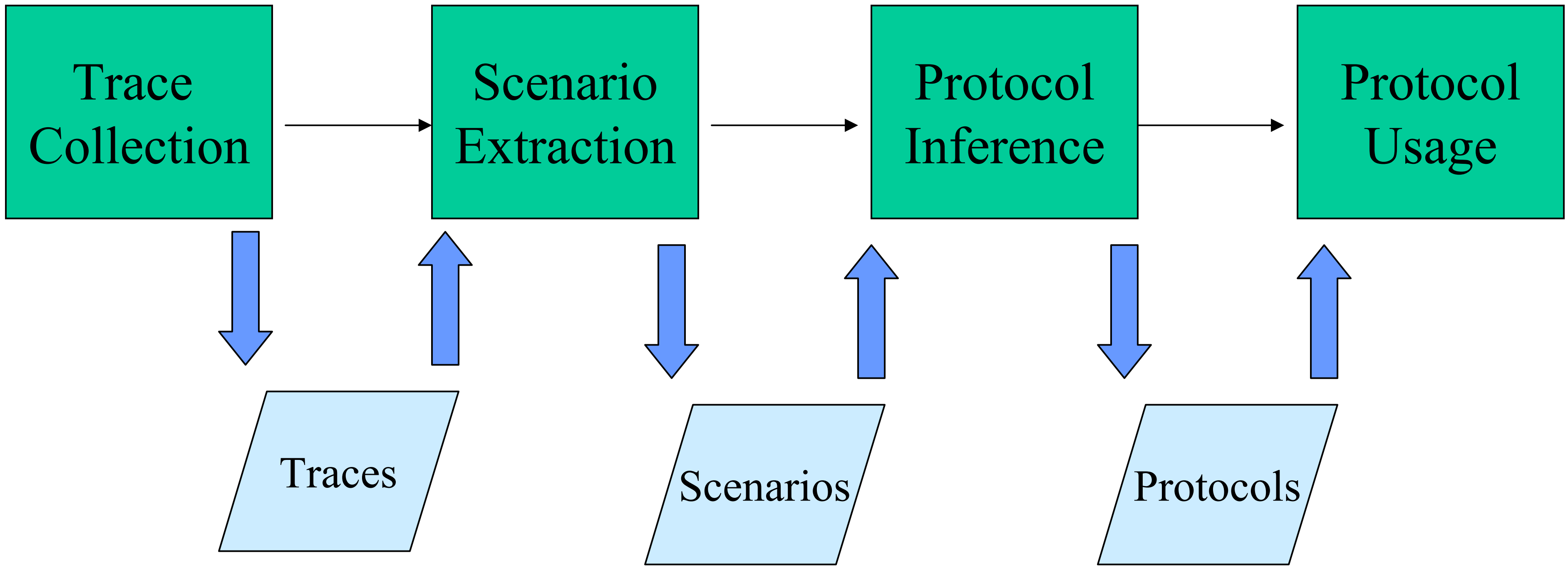
–Illegal sequences in traces or client code

–Method calls without any sequencing constraints

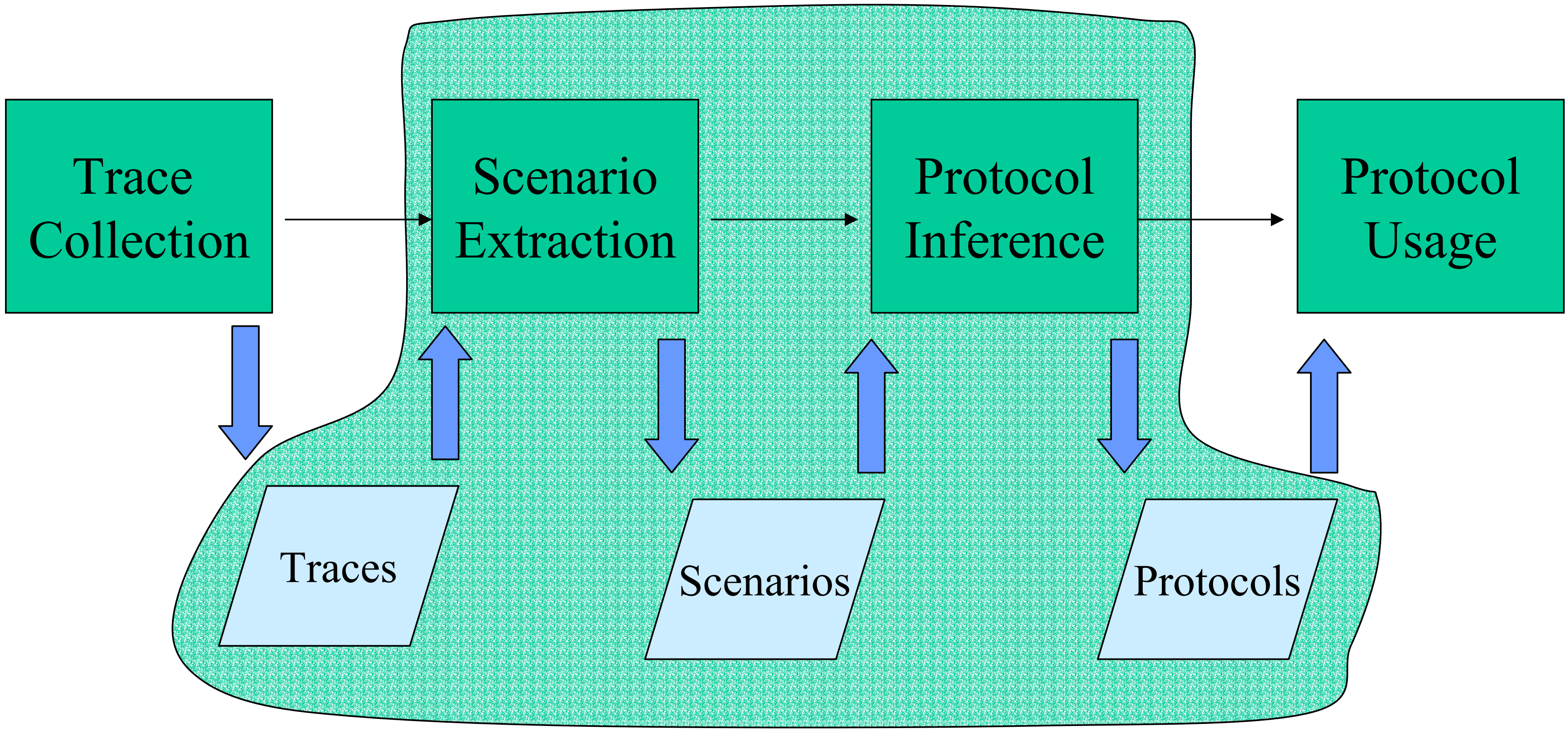
Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions

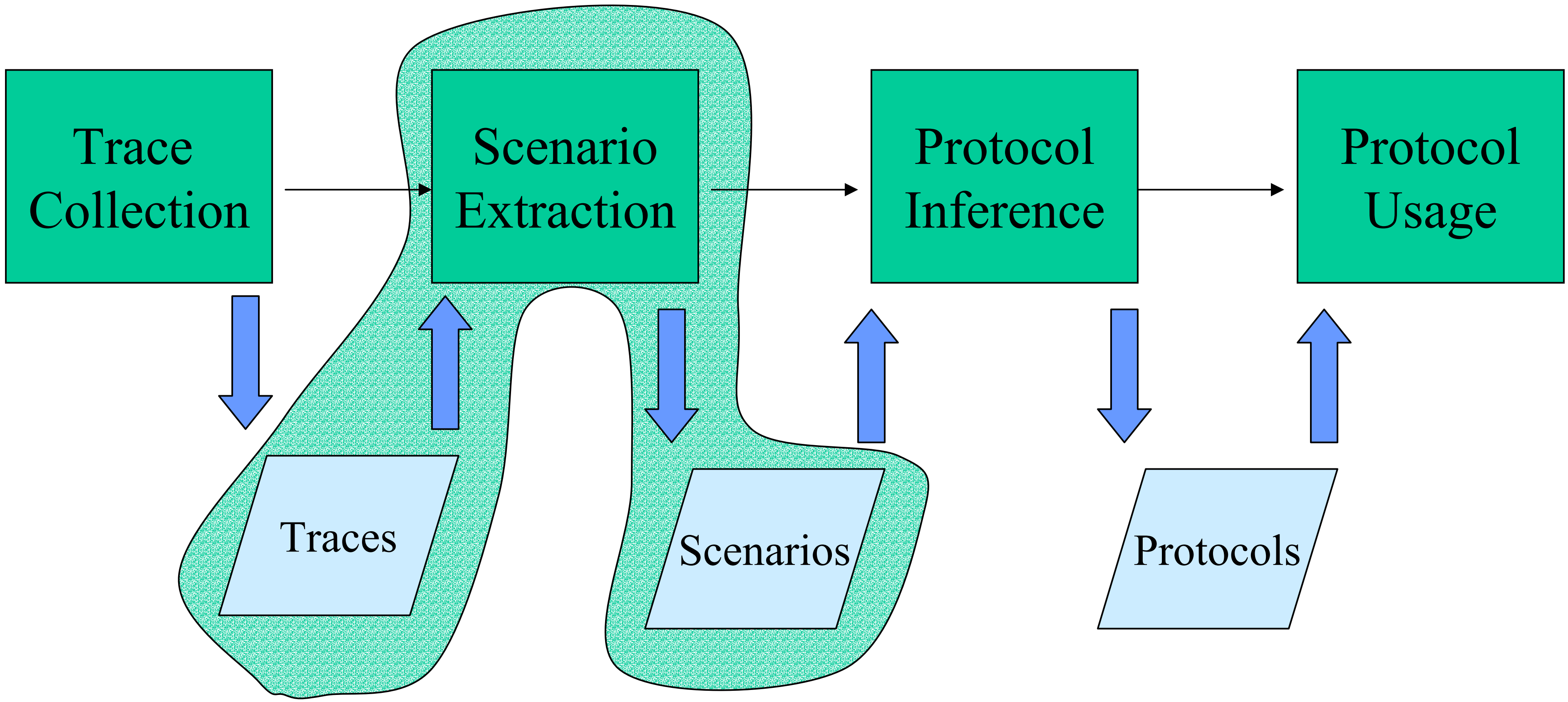
Dynamic Protocol Inference Framework



Dynamic Protocol Inference Framework



Dynamic Protocol Inference Framework



Scenario Extraction

A component usage scenario consists of **interdependent** method calls to a component interface

Why scenario extraction?

- Interleaving independent calls

Object1	Object2
	setMethod
	putNextEntry
	write
setMethod	
putNextEntry	
write	
closeEntry	
close	
	write
	closeEntry
	close

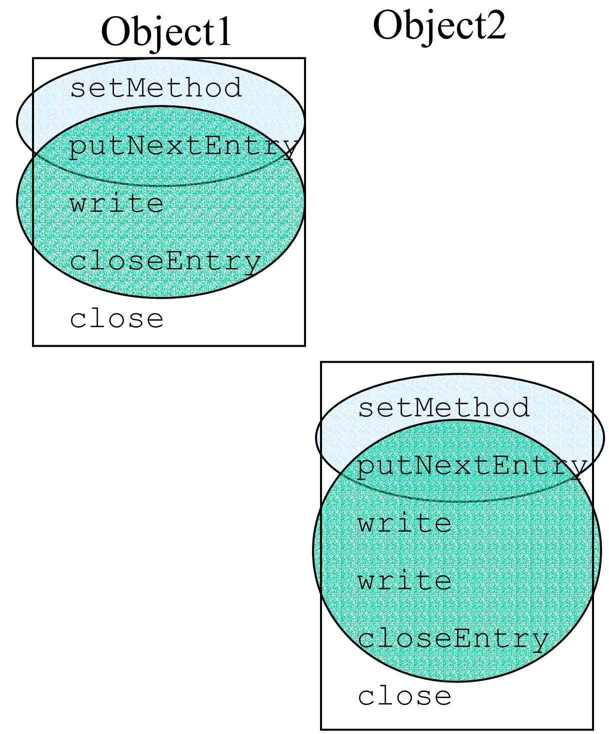
- Neighboring independent calls

Object1	Object2
setMethod	
putNextEntry	
write	
closeEntry	
close	
	setMethod
	putNextEntry
	write
	write
	closeEntry
	close

- OO program traces
- C program traces

Scenario Extraction from OO Program Traces

- Group by object [Reiss et al.]
  - –Method calls on **the same object**
  - –A single FSA model for a class
- Group by member fields [Whaley et al.]
  - –Method calls on **the same object**
  - –Method calls that access **the same field**
  - –*n* FSA submodels for a class with *n* fields

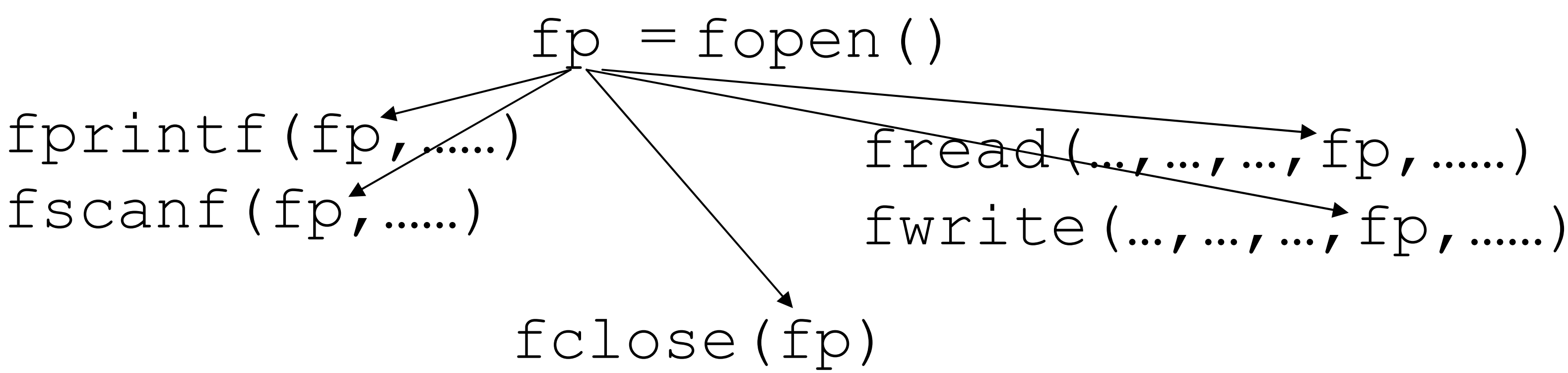


The `entry` field: `putNextEntry, write, closeEntry`

The `method` field: `setMethod, putNextEntry`

Scenario Extraction from C Program Traces-I

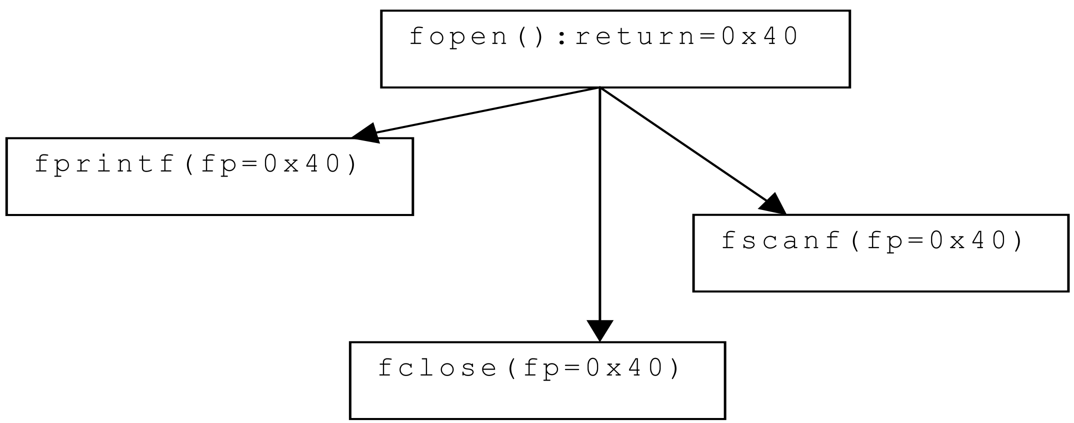
- Arguments and return values are used to group traces [Ammons et al.]



Scenario Extraction from C Program Traces-II

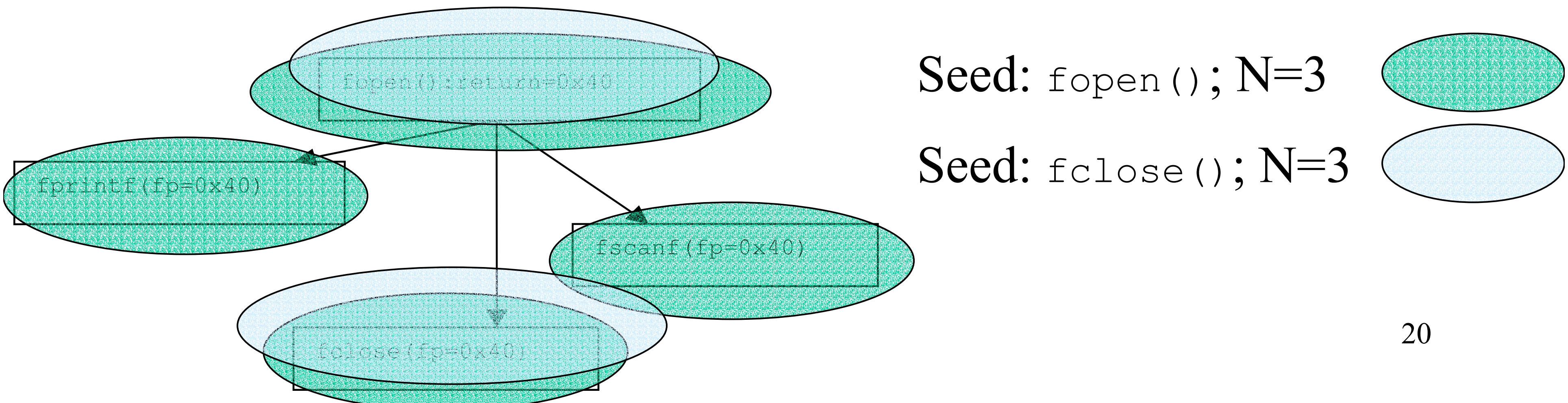
- User-specified attributes of an abstract object
  - Definers: `fopen.return; fclose.fp`
  - Users: `fprintf.fp; fscanf.fp; fclose.fp; fread.fp; fwrite.fp`
- Flow dependency analysis

```
fopen():return=0x40,fprintf(fp=0x40),fscanf(fp=0x40),fclose(fp=0x40)
```

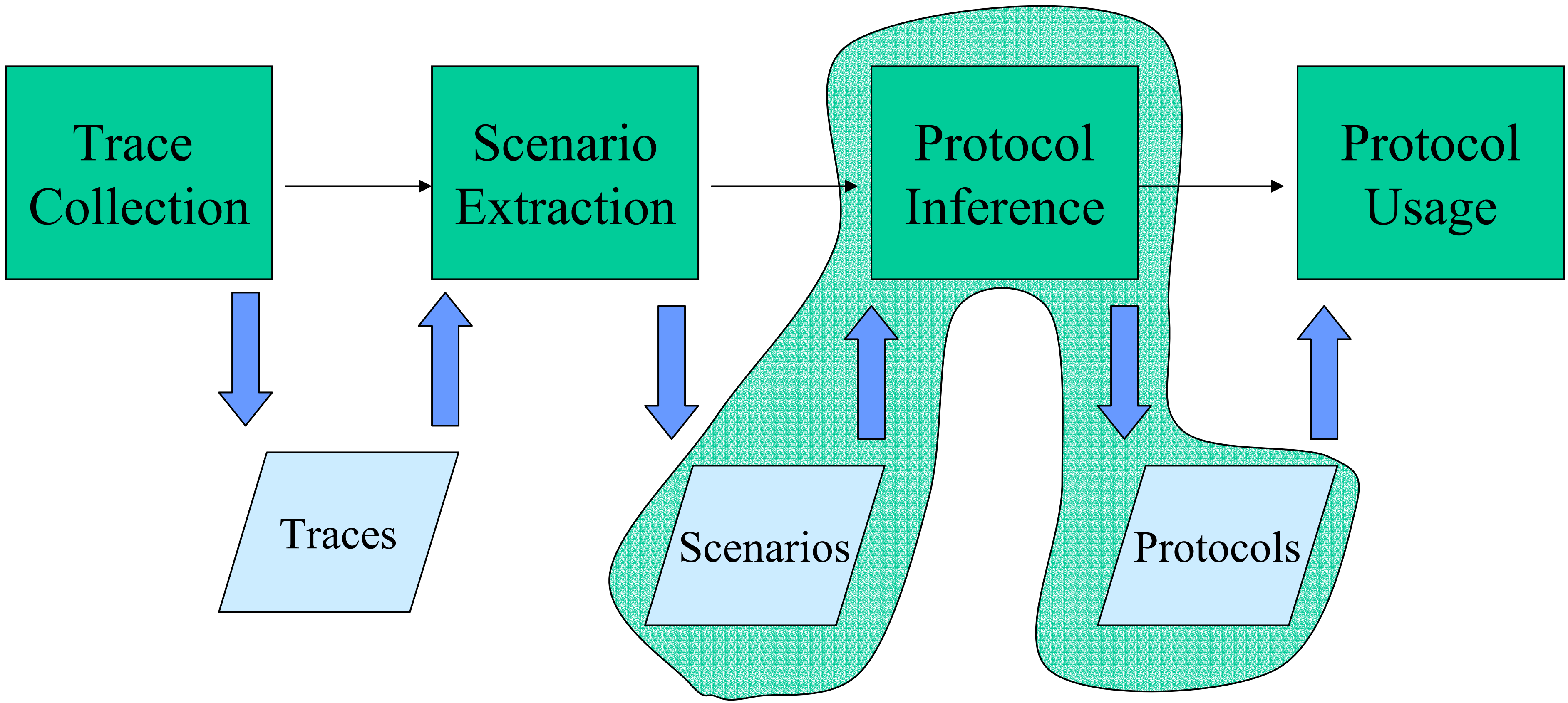


Scenario Extraction from C Program Traces-III

- A scenario is a set of function calls related by flow dependences.
  - User-specified scenario seeds and bounded size *N*
  - Scenario: ancestors and descendants of the seed function call



Dynamic Protocol Inference Framework



Protocol Inference

- A learning activity
  - Find a protocol
    - **explain** the given scenarios
    - **predict** future scenarios.
- Inputs: positive or negative scenarios
- Algorithms
  - *k*-tails Algorithm [Reiss et al][Ammons et al.][Cook et al.]
  - Separation of state-preserving methods [Whaley et al.]
  - Markov algorithm [Cook et al.]
  - IPM2 algorithm [El-Ramly et al.]

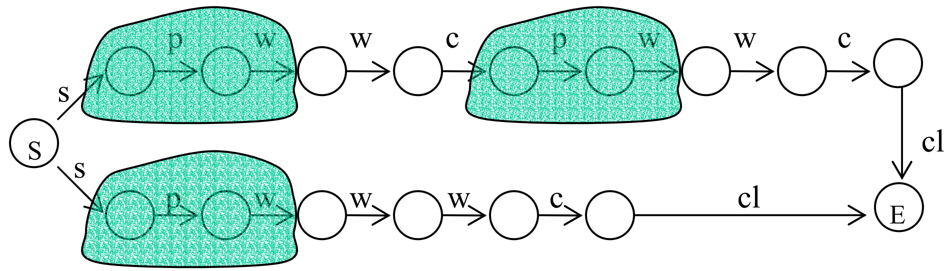
k-tails Algorithm [Biermann et al. 72]

- A state is defined by what future behavior can occur from it
  - The future (the *k*-tail): the next *k* method calls
  - Merge two states
    - if they have a *k*-tail in common [Reiss et al.]
    - if one includes all the *k*-tails of the other one [Cook et al.]

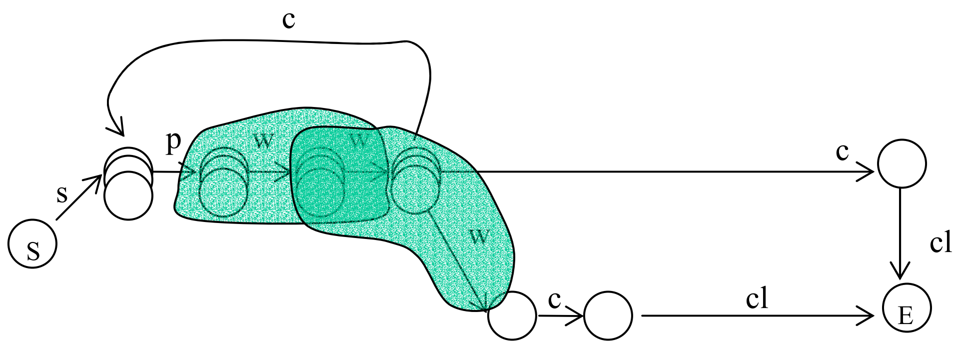
k-tails Algorithm Example (*k*=2 [Reiss et al.])

- `setMethod,putNextEntry,write,write,closeEntry,putNextEntry,write,write,closeEntry,close`
- `setMethod,putNextEntry,write,write,write,closeEntry,close`

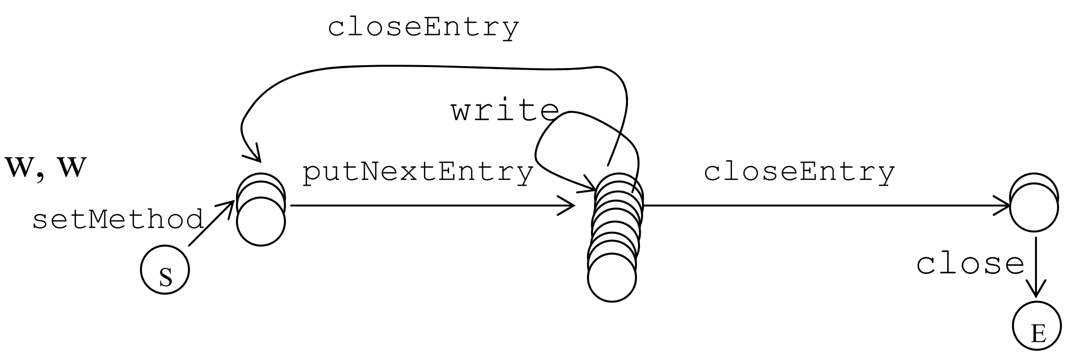
Initial FSA



Merge 2-tail of p, w



Merge 2-tail of w, w



Noise:

- States with low frequency [Cook et al.]
- Edges with low frequency [Ammons et al.]

Separation of State-Preserving Methods

[Whaley et al.]

- A submodel contains all the methods accessing the same field *f*.
  - e.g. `putNextEntry, write, closeEntry` (the `entry` field)

➢ State-modifying methods

–write *f;* change the object state

–e.g. `putNextEntry, closeEntry`

➢ State-preserving methods

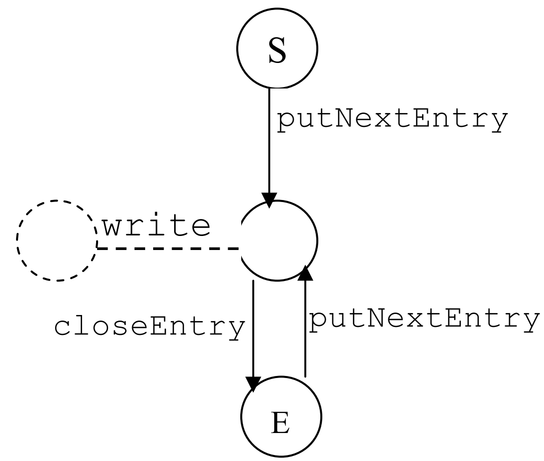
–only read *f;* not change the state of an object

–e.g. `write`

Submodel Extraction for the `entry` field

~~setMethod~~,putNextEntry,write,write,closeEntry,putNextEntry,write,write,closeEntry,~~close~~

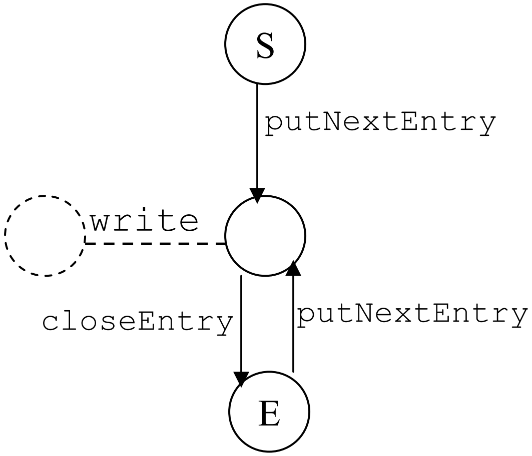
Last state-modifying method history	Method call
START	putNextEntry()
putNextEntry()	write()
putNextEntry()	write()
putNextEntry()	closeEntry()
closeEntry()	putNextEntry()
putNextEntry()	write()
putNextEntry()	write()
putNextEntry()	closeEntry()
closeEntry()	END



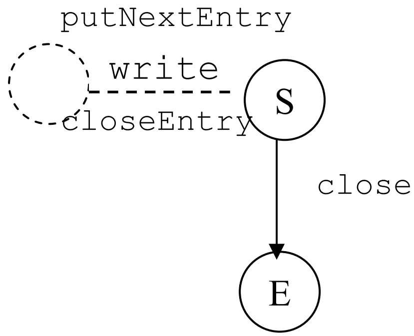
~~setMethod()~~,putNextEntry(),write(),write(),write(),closeEntry(),~~close()~~

Last state-modifying method	Method call
START	putNextEntry()
putNextEntry()	write()
putNextEntry()	write()
putNextEntry()	write()
putNextEntry()	closeEntry()
closeEntry()	END

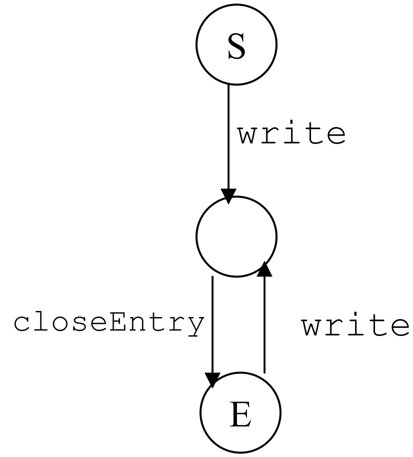
Submodels for `zipOutputStream`



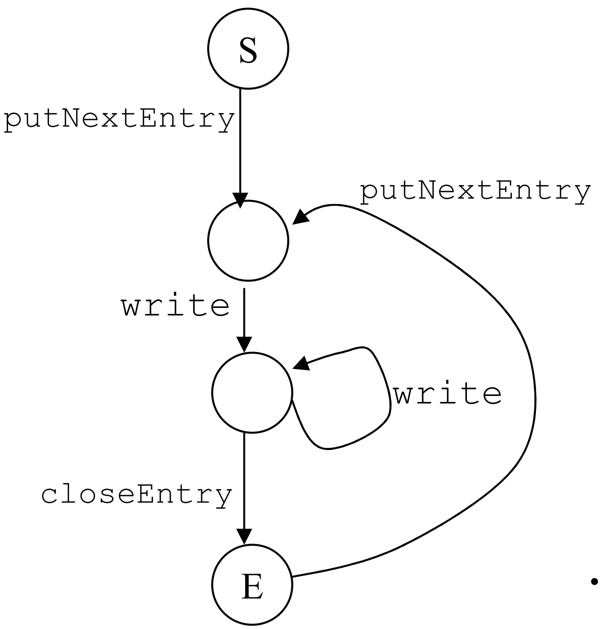
Submodel for the `entry` field



Submodel for the `closed` field

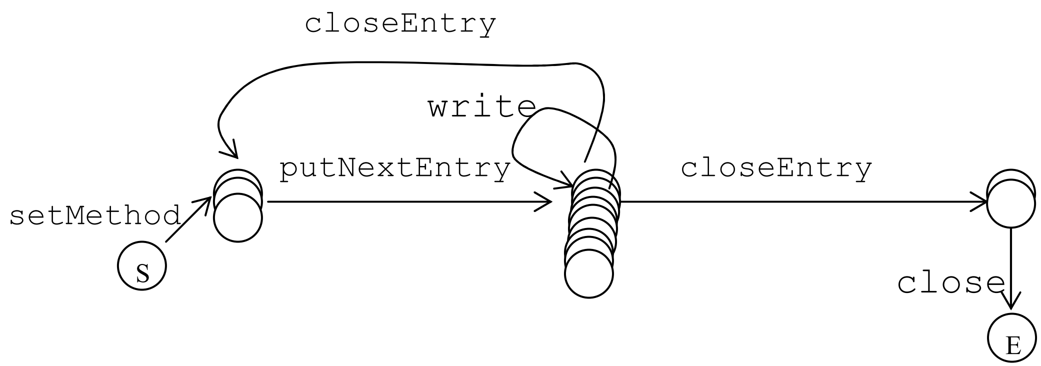


Submodel for the `crc` field



Submodel for the `written` field

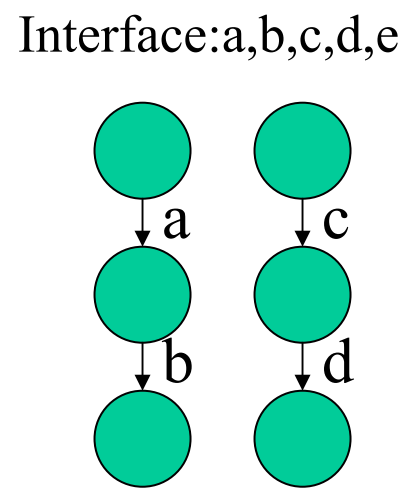
......



A single FSA model by 2-tails algorithm

Challenges Revisited

- Overgeneralization/over-restrictiveness
  - Overgeneralization: accept some illegal sequences
  - Over-restrictiveness: reject some legal sequences



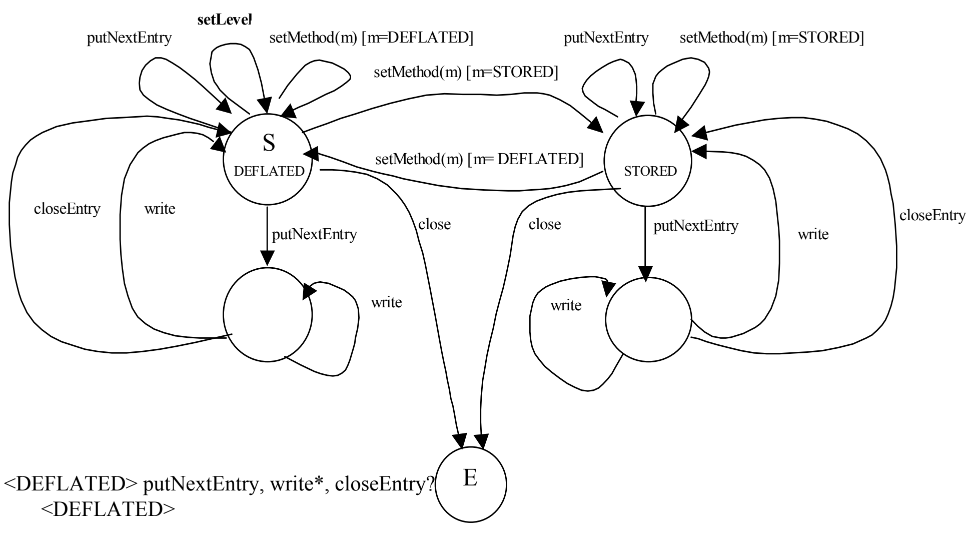
- Separation/composition of constraints
  - e.g. `DEFLATED` and `STORED` groups
  - e.g. Concurrent FSAs

- Data-dependent transitions
  - e.g. `setMethod(`**`DEFLATED`**`),setMethod(`**`STORED`**`)`
  - e.g. `pop()` when `currentSize>0`

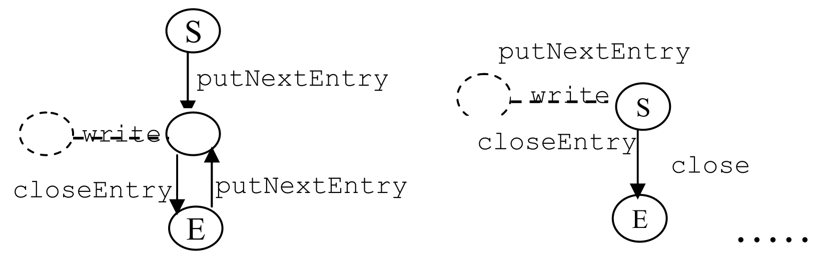
- Robustness to noise
  - Illegal sequences in traces or client code
  - Method calls without any sequencing constraints

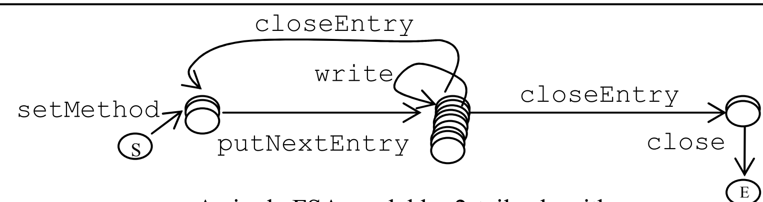
Challenges Revisited

Previous work	**overgeneralization/ over-restrictiveness**	separation/ composition of constraints	data-dependent transitions	robustness to noise
Whaley et al.	×	Separation	×	Handling unrelated methods by separation
Reiss et al.	×	Composition	×	×
Ammons et al.	×	Composition	×	Removing edges with low frequency
Cook et al.	×	Composition	×	Removing states with low frequency



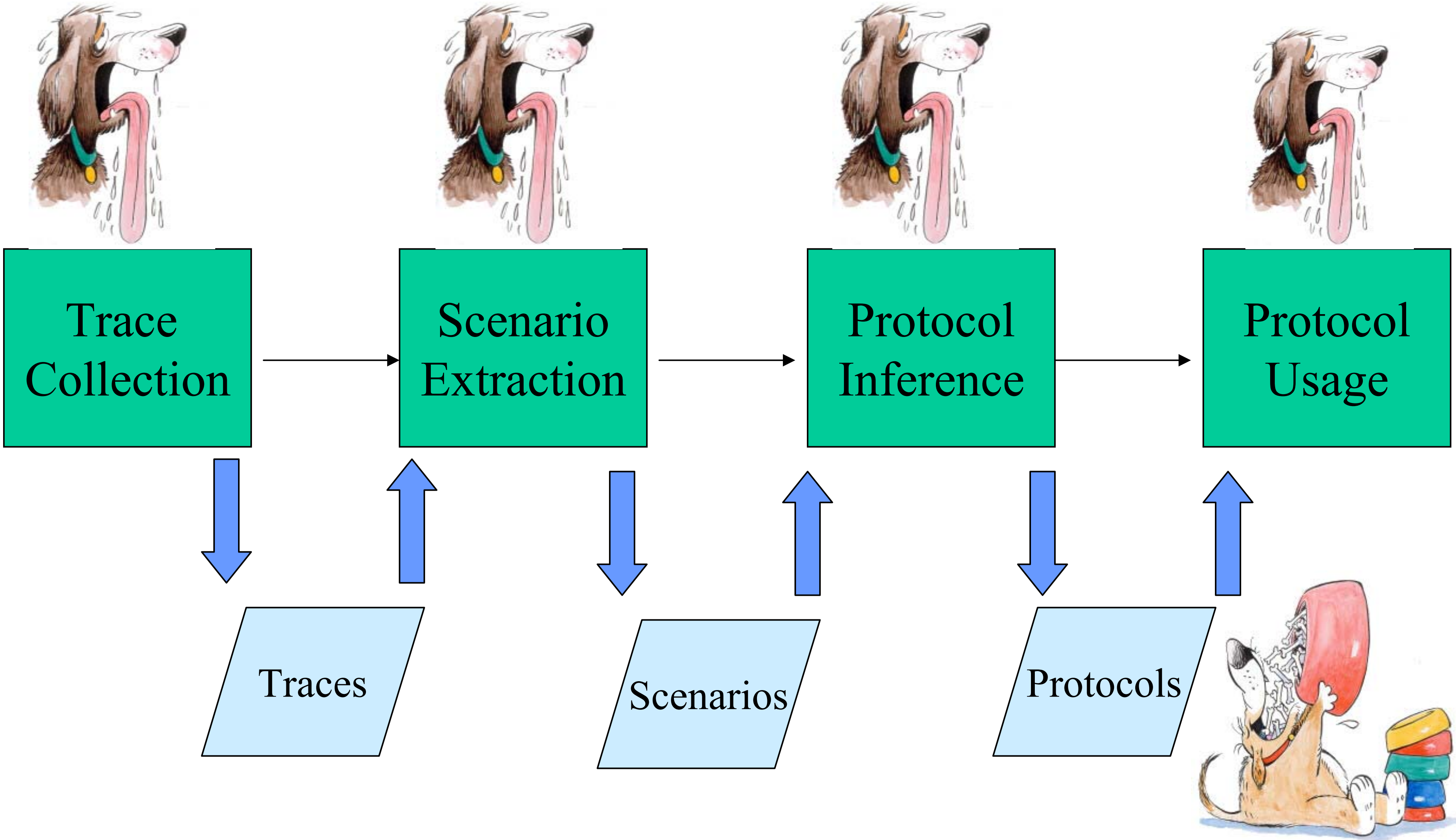
Submodel for the `entry` field   Submodel for the `closed` field





A single FSA model by 2-tails algorithm

Dynamic Protocol Inference Framework



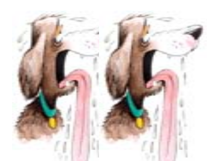
Evaluation: Cost-Benefit Analysis

Cost-Benefit Analysis - Cost

- Trace collection
  - Analysis scope [Ammons et al.][Cook et al.][Reiss et al.][Whaley et al.]
- Scenario extraction
  - Abstract object attributes [Ammons et al.]
  - Scenario seeds [Ammons et al.]
  - Scenario bounded size *N* [Ammons et al.]
- Protocol inference
  - Algorithm parameters [Ammons et al.][Cook et al.][Reiss et al.]
  - Noise thresholds [Ammons et al.][Cook et al.]
- Protocol usage 

Ammons et al.

Cook et al.



Reiss et al.

Whaley et al.

Cost-Benefit Analysis - Benefit

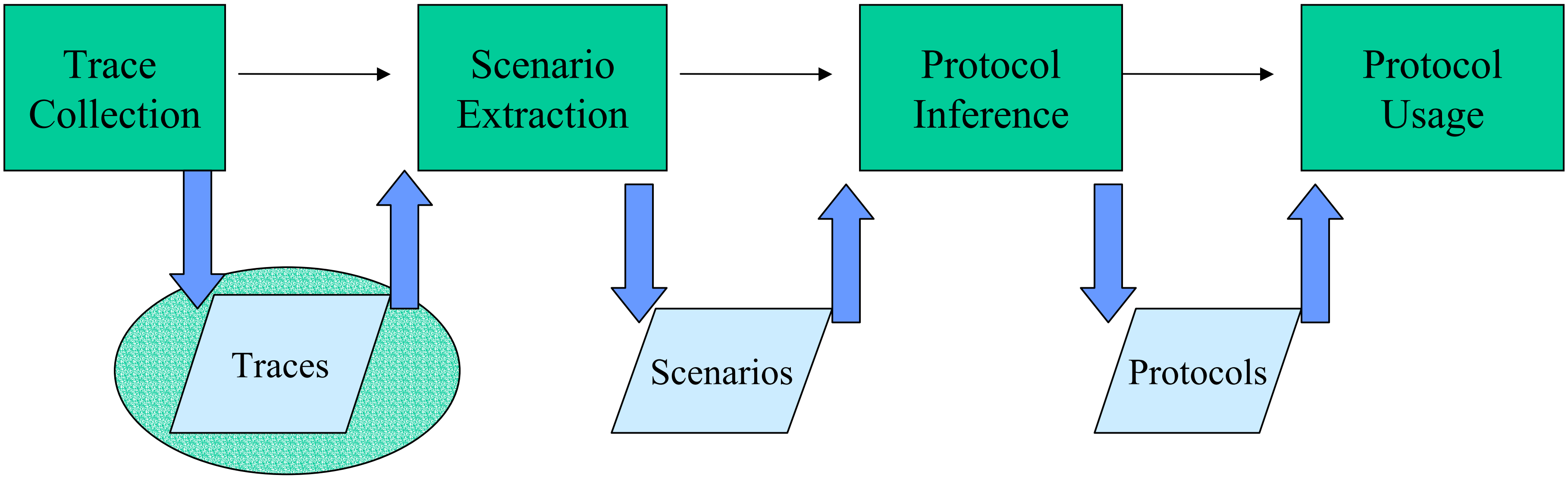
- Accuracy
- Usefulness in particular applications
- Case studies
  - Whaley et al.
    - J2EE (50 "*very interesting*" models/657 classes)
    - 1 method in joeq program
  - Ammons et al.
    - 1 documented rule for *X11* windowing sys (2000 functions)
    - 17 *X11* clients (96 scenarios), 5 violating programs (2 buggy)
    - 72 clients (90 traces), 17 inferred "*useful*" specs, 2/3 detect 199 true bugs [Ammons 03]
  - Cook et al.
    - A change request process, 159 traces* 32 events, reflect 65% vs. 40%

Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions

Static Protocol Inference Techniques

- Static analysis of client code [Lie et al. 01]



- Static analysis of component code [Whaley et al.]

Static Analysis of Component Code [Whaley et al.]

– Defensive programming

```
public void closeEntry() throws IOException {
    ……
    entry = null;
}

public synchronized void write(byte[] b, int off, int len) throws IOException {
    ……(no writes of entry)

    if (entry == null) {
        throw new ZipException("no current ZIP entry");
    }
    ……
}
```

- `closeEntry()`, `write()` is not allowed

- Select exception-guarding predicates and related fields in *m*
- Find method *m'* to set the fields to constants
- Identify illegal sequences from *m* to *m'*

Experimental results:

Java standard class library (81/914 classes, 24 listed)

Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
  - Component testing
  - Inference improvement
- Conclusions

Component testing-I

Negative samples from component tests

- Component tests provide negative samples
  - Test case: `write, putNextEntry`
- Automatic test generation for a submodel
  - Submodel for the `entry` field:

    `putNextEntry, write, closeEntry`

    Generate call sequences:

    `putNextEntry, write` √

    `write, putNextEntry` ×

    `putNextEntry, closeEntry` √

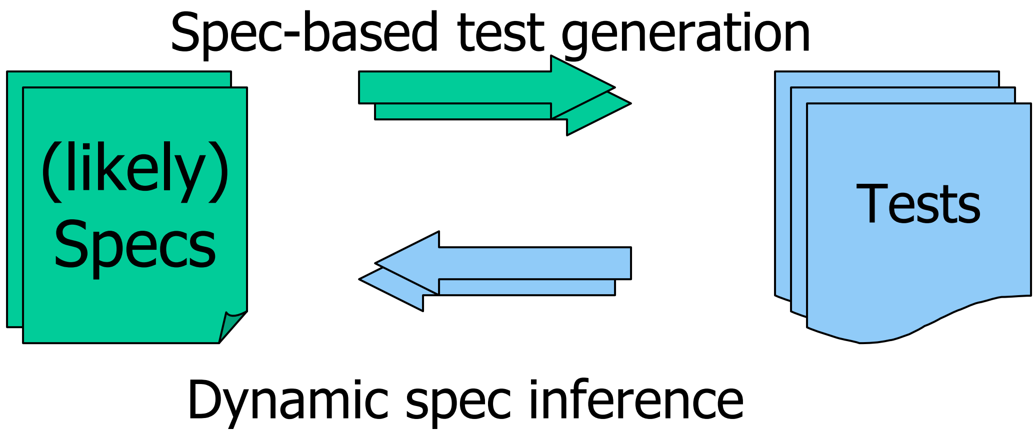
    `closeEntry, putNextEntry` √

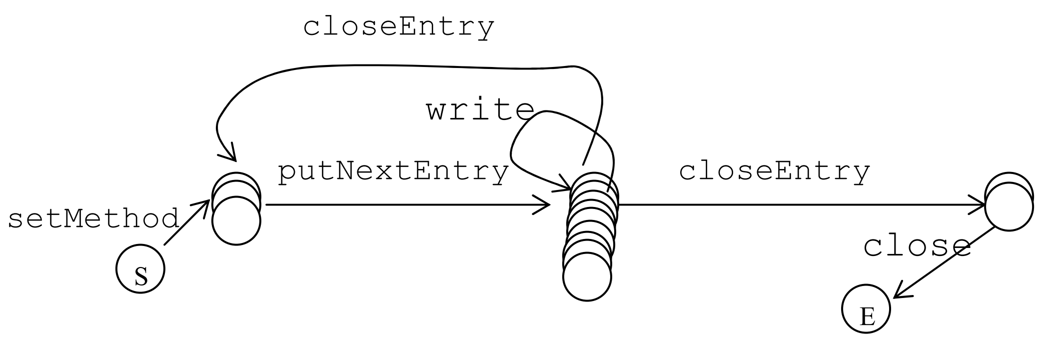
    `write, closeEntry` ×

    `closeEntry, write` ×

Component testing-II
Feedback loop between component testing and protocol inference

- Better protocols ←→ better tests



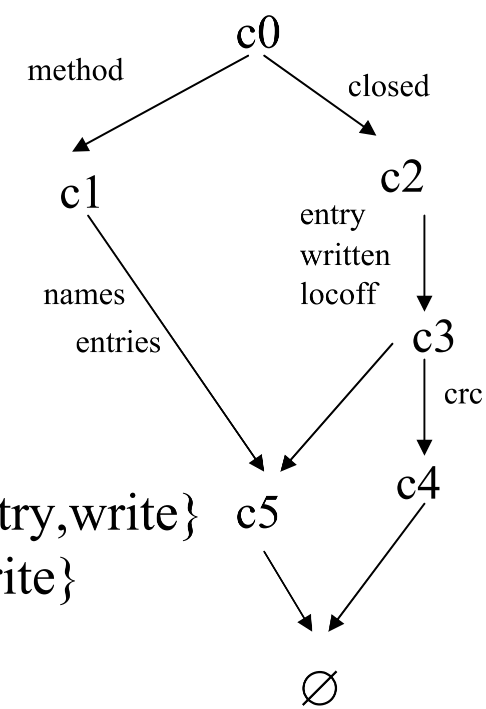


A single FSA model by 2-tails algorithm

Inference Improvement-I
Composition and separation of constraints

- Concept analysis [Wille 82] to compose constraints

methods \ fields	entry	entries	crc	written	locoff	closed	method	names
putnextEntry	W	W		W	W	R	R	W
write	R		W	W	R	R		
closeEntry	W		W	W	R	R		
close						W		
setMethod							W	



c0=all methods
c1={putnextEntry,setMethod}
c2={close,closeEntry,putnextEntry,write}
c3={closeEntry,putnextEntry,write}
c4={closeEntry,write}
c5={putnextEntry}

- Cluster analysis [Anderberg 73] to separate constraints

Inference Improvement-II

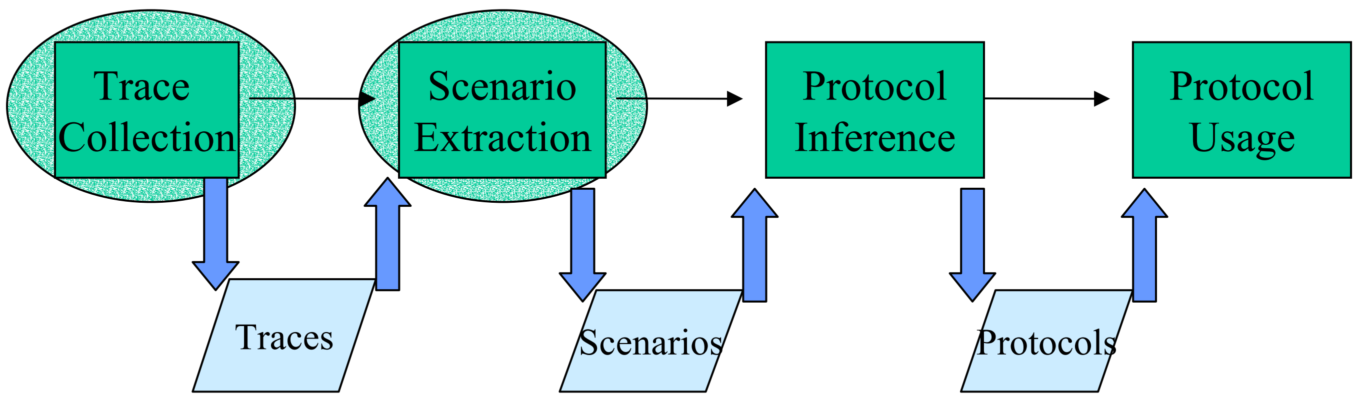
Data-dependent transition inference

- Data-dependent transitions
  - e.g. `setMethod(`**`DEFLATED`**`),setMethod(`**`STORED`**`)`
  - e.g. `pop()` when `currentSize>0`

- Heuristics to identify the data related to a component mode
  - Side-effect-free boolean methods
    - `isEmpty()`, `isFull()` in `Stack` class
  - Member fields in conditionals
    - `if (currentSize>0)`, `if (currentSize==MAXSIZE)`
    - `switch (method)`
      `{ case DEFLATED:… case STORED:… }`

New Problem: Argument Object Sequencing Constraint Inference

- Problem: before calling `putNextEntry(ZipEntry e)` with argument `e`,
  - What method calls in `ZipEntry` need to be invoked on object `e`?
  - What method calls in `ZipOutputStream` need to be invoked by passing `e`?
- Related to bi-directional protocols for collaboration



Outline

- Background
- Overview of protocol inference
- Dynamic protocol inference framework
- Static protocol inference techniques
- Future work
- Conclusions

Conclusions

- Discussed component protocol inference problems and identified challenges
- Proposed a dynamic inference framework to compare previous work
- Discussed static inference techniques
- Suggested future work in the area

Trace Collection - I

Collected data types for a method call

- Method signature. [Whaley et al.][Reiss et al.][Ammons et al.]
  - Software process [Cook et al.]
  - Screen ID [El-Ramly et al.]
- Sequencing order (all)
- Class/Object ID [Whaley et al.][Reiss et al.] or arguments and return values [Ammons et al.]

Trace Collection - II

Summary of data collection mechanisms

Previous work	Source code instrumentation	Bytecode/executable instrumentation	Execution environment
Whaley et al.		√ (component code)	
Reiss et al.			√ (JVMPI)
Ammons et al.		√ (client code)	
Cook et al.	N/A	N/A	N/A
El-Ramly et al.	N/A	N/A	N/A

Trace Collection - III

Comparison of data collection mechanisms

- Component code instrumentation [Whaley et al.]
  - + does it once for all (clients)
  - + without requiring the availability of the client code
- Client code instrumentation [Ammons et al.]
  - + better control of the instrumentation scope
  - + without requiring the availability of the component code
- Execution environment using Java Virtual Machine Profiling Agent (JVMPI) [Reiss et al.]
  - + Combine the above two

Trace Collection Internal usage of component

- Methods in the interface are called by component itself
- Internal usage needs to be identified and filtered out
  - Whaley et al. maintain knowledge of the local call stack
  - Reiss et al. post-process the collected traces.

```
public void putNextEntry(ZipEntry e) throws IOException {
        ensureOpen();
        if (entry != null) {
            closeEntry();// close previous entry
        }
        ......
}
```

Online vs. Offline Analysis

- Online analysis - Whaley et al.
  - Performed while the system is running
- Offline analysis- Reiss et al., Ammons et al., Cook et al., and El-Ramly et al.
  - Performed after the system has terminated

IPM2 algorithm [El-Ramly et al.]

- Given two scenarios: 1,3,2,3,4,3 and 2,3,2,4,1,3
- Infer two patterns: 2,3,4 and 3,2,4,3

1,3,2,3,4,3	1,3,2,3,4,3
2,3,2,4,1,3	2,3,2,4,1,3

Protocol Usage

- Without tool supports
  - Characterizing test suite [Whaley et al.]
  - Understanding systems [Whaley et al.]
  - Assisting spec construction [Whaley et al.]
  - Tuning algorithm parameters [Reiss et al.]
- With tool supports
  - Auditing applications [Whaley et al.]
  - Debugging specifications [Ammons et al.]

Summary of Dynamic Inference Techniques

Previous work	Trace collection	Scenario extraction	Protocol inference	Protocol usage
Whaley et al.	Method calls, Class/Object Ids	Object-based, Slicing by member fields	Separation of state modifying and state preserving methods	Test suite characterization, Software auditing
Reiss et al.	Method calls, Class/Object Ids	Object-based	k-tails algorithm	Alg parameter tuning
Ammons et al.	Method calls, Argument/return values	Flow dependence, Simplification, Standardization	sk-strings algorithm	Trace verification, Specification debugging
Cook et al.	Process events	n/a	k-tails algorithm, Markov algorithm	Process validation
El-Ramly et al.	Screen Ids	Interaction-based	IPM2 algorithm	Legacy system reengineering

Static Analysis of Client Code

- Scenarios can be extracted from code statically as inputs to protocol inference algorithms.
  - Model checking:
    - models extracted from code by using pattern matching and program slicing [Lie et al. 01].
  - Intrusion detection
    - an FSA for system calls inferred from application code [Wagner et al. 01].
  - Bug detection
    - temporal rules inferred from the Linux code [Engler et al. 01]