The CLOSER: Automating Resource Management in Java

Isil Dillig       Thomas Dillig       Eran Yahav       Satish Chandra
Computer Science Department       Stanford University       IBM T.J. Watson Research Center

ISMM 2008
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
  - Operating system resources: Files, sockets, ...
Motivation

Operating System Resources

```java
public void transferData()
{
    Socket s = new Socket();
    s.connect(...);
    ...
    s.close();
}
```
motivation

operating system resources

```java
public void transferData()
{
    Socket s = new Socket();
    s.connect(...);
    ...
    s.close();
}
```
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
  - Operating system resources: Files, sockets, ...
  - Window system resources: Fonts, colors, ...
Motivation

Window System Resources

```java
public void draw()
{
    Font f = new Font();
    ...
    f.dispose();
}
```
Motivation

Window System Resources

```java
public void draw()
{
  Font f = new Font();
  ...
  f.dispose();
}
```
Motivation

- Automatic garbage collection in Java has relieved programmers from the burden of manual memory management.

- Unfortunately, memory is not the only resource.
  - Operating system resources: Files, sockets, ...
  - Window system resources: Fonts, colors, ...
  - Application specific resources: Listeners, model view control pattern, ...
Motivation

Application Specific Resources

```java
public class SomeView {
    private SomeListener l;
    private WorkbenchWindow w;

    public void createPartControl(Composite parent) {
        l = new Listener(this);
        w.addPerspectiveListener(l);
    }

    public void dispose() {
        w.removePerspectiveListener(l);
    }
}
```
Motivation

Application Specific Resources

```java
public class SomeView {
    private SomeListener l;
    private WorkbenchWindow w;

    public void createPartControl(Composite parent) {
        l = new Listener(this);
        w.addPerspectiveListener(l);
    }

    public void dispose() {
        w.removePerspectiveListener(l);
    }
}
```
Definition of a Resource

A resource $r$ is an instance of any type whose specification has the following requirement:
A resource $r$ is an instance of any type whose specification has the following requirement:

- If a method $m$ is called with $r$ as the receiver or parameter
## Generalized Definition of Resource

### Definition of a Resource

A resource $r$ is an instance of any type whose specification has the following requirement:

- If a method $m$ is called with $r$ as the receiver or parameter
- Then a matching method $m'$ must be called after the last use of $r$. 

---
A resource \( r \) is an instance of any type whose specification has the following requirement:

- If a method \( m \) is called with \( r \) as the receiver or parameter
- Then a matching method \( m' \) must be called after the last use of \( r \).

We call \( m \) the **obligating** method and \( m' \) the **fulfilling** method.
Existing Approaches and Their Drawbacks

- Manual Resource Management
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...
Existing Approaches and Their Drawbacks
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ... 

- Finalization
Existing Approaches and Their Drawbacks

- **Manual Resource Management**
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- **Finalization**
  - In current JVM implementations, program might run out of non-memory resources before finalizers are called
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- Finalization
  - In current JVM implementations, program might run out of non-memory resources before finalizers are called
  - Asynchronous with respect to last use point
Existing Approaches and Their Drawbacks

- Manual Resource Management
  - Same drawbacks as manual memory management: leaks, double disposes, ...

- Finalization
  - In current JVM implementations, program might run out of non-memory resources before finalizers are called
  - Asynchronous with respect to last use point
  - And therefore almost never used in practice
What is Ideal Resource Management?

- Dispose resource after its last use (read or write).
Is This Really "Ideal Resource Management"?
Is This Really "Ideal Resource Management"?
Is This Really "Ideal Resource Management"?
What is Ideal Resource Management?

- Dispose resource after its last relevant use.
What is Ideal Resource Management?

- **Dispose resource after its last relevant use.**
  - Unfortunately, determining last use is impossible to do dynamically and difficult to approximate statically, especially in the case of open programs.
What is Ideal Resource Management?

- **Dispose resource after its last relevant use.**
  - Unfortunately, determining last use is impossible to do dynamically and difficult to approximate statically, especially in the case of open programs.

- **Solution:** Just as last use is approximated by traditional notion of reachability, we approximate last relevant use by interest reachability.
Interest Reachability

- Differentiate between interest and non-interest links.
Interest Reachability

- Differentiate between interest and non-interest links.
  - If A references B through a non-interest link, then the relevant behavior of A does not depend on the existence of B.
Interest Reachability

- Differentiate between interest and non-interest links.
  - If A references B through a non-interest link, then the relevant behavior of A does not depend on the existence of B.
  - Non-interest links must be annotated by the programmer since "relevant" behavior defines application semantics.
Our Goal

We guarantee that a resource is disposed as soon as it becomes unreachable through interest links.
Our Goal

We guarantee that a resource is disposed \textit{as soon as} it becomes unreachable through interest links.

- Advantages:
Our Goal

We guarantee that a resource is disposed as soon as it becomes unreachable through interest links.

Advantages:

- Resource drag is much shorter compared to asynchronous approaches.
Our Goal

We guarantee that a resource is disposed as soon as it becomes unreachable through interest links.

Advantages:

- Resource drag is much shorter compared to asynchronous approaches.

- Works even if disposing the resource has visible side effect (e.g., disposal removes button from a window).
Interest Reachability

Observer

Listener

Observed
Interest Reachability
Interest Reachability

Listener ➔ Observed
Interest Reachability

```java
o.removeListener(l)
```
Interest Reachability

Listener

Observed
How to Achieve this Goal

Recall:

We want to guarantee that a resource is disposed as soon as it becomes unreachable through interest links.
How to Achieve this Goal

To achieve this goal:

Whenever possible, statically identify the first program point where resource becomes unreachable through interest links.

When this is not possible, identify the correct dispose point using a variation of reference counting.
How to Achieve this Goal

To achieve this goal:

- Whenever possible, statically identify the first program point where resource becomes unreachable through interest links.
How to Achieve this Goal

To achieve this goal:

- Whenever possible, statically identify the first program point where resource becomes unreachable through interest links.
- When this is not possible, identify the correct dispose point using a variation of reference counting.
Problem: Resource Sharing

A Font object is shared between two Window objects and should be disposed when last window is closed by the user:
Overview of Our Approach

- The user annotates:
  - the set of **primitive resources**
Overview of Our Approach

class WorkbenchWindow {

    private Listener l;

    @Obligation(obligates = 'removePerspectiveListener',
                resource=1)
    public void addPerspectiveListener(Listener l);
    ...

}
Overview of Our Approach

class WorkbenchWindow {

    private Listener l;

    @Obligation(obligates = ‘removePerspectiveListener’,
                resource=1)
    public void addPerspectiveListener(Listener l);

    ...

}
Overview of Our Approach

class WorkbenchWindow {

    private Listener l;

    @Obligation(obligates = 'removePerspectiveListener',
                resource=1)
    public void addPerspectiveListener(Listener l);
    ...

}
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of **non-interest-links**
Overview of Our Approach

class WorkbenchWindow {

    @NonInterest
    private Listener l;

    @Obligation(obligates = "removePerspectiveListener", resource=1)
    public void addPerspectiveListener(Listener l);
    ...
}

Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
  - and later automatically synthesizes dispose methods.
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
  - and later automatically synthesizes dispose methods.

- CLOSER statically analyzes resource lifetimes to identify how and where each resource should be disposed.
Overview of Our Approach

- The user annotates:
  - the set of primitive resources
  - the set of non-interest-links

- CLOSER infers:
  - the set of higher-level resources
  - and later automatically synthesizes dispose methods.

- CLOSER statically analyzes resource lifetimes to identify how and where each resource should be disposed.

- CLOSER automatically inserts any appropriate resource dispose calls into source code.
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the resource interest graph (RIG).
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the **resource interest graph (RIG)**.

### Resource Interest Graph

An RIG for a method $m$ at a given point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where:

- $V$ is a finite set of abstract memory locations
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the **resource interest graph (RIG)**.

**Resource Interest Graph**

An RIG for a method \( m \) at a given point is a tuple \( \langle V, E, \sigma_V, \sigma_E \rangle \) where:

- \( V \) is a finite set of abstract memory locations
- \( E \) is a set of directed edges between these locations
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the resource interest graph (RIG).

**Resource Interest Graph**

An RIG for a method $m$ at a given point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where:

- $V$ is a finite set of abstract memory locations
- $E$ is a set of directed edges between these locations
- $\sigma_V$ is a mapping from abstract memory locations to a value in 3-valued logic, identifying whether that location may, must, or must-not be a resource
To effectively reason about resource lifetimes, CLOSER utilizes a novel flow-sensitive points-to graph, called the resource interest graph (RIG).

An RIG for a method $m$ at a given point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where:

- $V$ is a finite set of abstract memory locations
- $E$ is a set of directed edges between these locations
- $\sigma_V$ is a mapping from abstract memory locations to a value in 3-valued logic, identifying whether that location may, must, or must-not be a resource
- $\sigma_E$ is a mapping from edges to a boolean value identifying whether that edge is an interest or non-interest edge
Example RIG

```java
public class BufferPrinter {
    ...
    public BufferPrinter(Buffer buf) {
        this.buf = buf;
        this.listener =
            new BufferListener(this);
        buf.addListener(listener);
        this.socket = new Socket();
        socket.connect();
    }
}
```
public class BufferPrinter {
    ...
    public BufferPrinter(Buffer buf) {
        this.buf = buf;
        this.listener =
            new BufferListener(this);
        buf.addListener(listener);
        this.socket = new Socket();
        socket.connect();
    }
}

\[
\sigma_v(A) = ?
\]

\[
\sigma_v(B) = 1
\]

\[
\sigma_v(C) = 1
\]

\[
\sigma_v(D) = ?
\]
A class $T$ is a higher-level resource if:

\[ \sigma_{V}(l_{T}) \subseteq \sigma_{E}(l_{T} \times f \to l_{f}) = \text{true} \]

If $T$ is inferred to be a higher-level resource, $T$'s constructor becomes an obligating method and the dispose method synthesized by CLOSER becomes the corresponding fulfilling method.
A class $\mathcal{T}$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $\mathcal{T}$
A class $\mathcal{T}$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $\mathcal{T}$
- such that $\sigma_V(l_f) \sqsubseteq 1$
A class $\mathcal{T}$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $\mathcal{T}$
- such that $\sigma_V(l_f) \supseteq 1$
- $\sigma_E(l_T \times f \rightarrow l_f) = \text{true}$
A class $T$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $T$
- such that $\sigma_V(l_f) \supseteq 1$
- $\sigma_E(l_T \times f \rightarrow l_f) = \text{true}$

If $T$ is inferred to be a higher-level resource,
A class $T$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $T$
- such that $\sigma_V(l_f) \supseteq 1$
- $\sigma_E(l_T \times f \rightarrow l_f) = true$

If $T$ is inferred to be a higher-level resource,

- $T$’s constructor becomes an obligating method
A class $\mathcal{T}$ is a higher-level resource if:

- there exists a field $l_f$ of some instance of $\mathcal{T}$
- such that $\sigma_V(l_f) \equiv 1$
- $\sigma_E(l_T \times f \to l_f) = true$

If $\mathcal{T}$ is inferred to be a higher-level resource,

- $\mathcal{T}$’s constructor becomes an obligating method
- and the dispose method synthesized by CLOSER becomes the corresponding fulfilling method.
Higher-Level Resource Example

\[
\begin{align*}
\sigma_E(e) &= 1 \\
\sigma_E(e) &= 0
\end{align*}
\]

\[
\begin{align*}
\sigma_V(A) &= 1 \\
\sigma_V(B) &= 1 \\
\sigma_V(C) &= 1 \\
\sigma_V(D) &= 0
\end{align*}
\]
Higher-Level Resource Example

\[
\begin{align*}
\sigma_E(e) &= 1 \\
\sigma_E(e) &= 0 \\
\sigma_v(A) &= 1 \\
\sigma_v(B) &= 1 \\
\sigma_v(C) &= 1 \\
\sigma_v(D) &= 0
\end{align*}
\]
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource's obligating method was called before disposing it.

- **Dynamic dispose**
  - Requires keeping a run-time "interest-count" needed whenever CLOSER infers that resource may be shared.
CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource’s obligating method was called before disposing it.
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource’s obligating method was called before disposing it.

- **Dynamic dispose**
Resource Disposal Strategies

CLOSER disposes of a resource in one of three ways:

- **Strong static dispose**
  - Dispose resource directly by calling fulfilling method
  - No checks necessary

- **Weak (conditional) static dispose**
  - Checks whether the resource’s obligating method was called before disposing it.

- **Dynamic dispose**
  - Requires keeping a run-time “interest-count”
  - Needed whenever CLOSER infers that resource may be shared.
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique solicitor for it.
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique
  **solicitor** for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to
dispose $r$. 
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique solicitor for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$. 
CLOSER proves a resource is unshared if it can identify a unique **solicitor** for it.

If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$. 
Solicitors

- CLOSER proves a resource is unshared if it can identify a unique solicitor for it.
- If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$.
- CLOSER infers a solicitor by:
CLOSER proves a resource is unshared if it can identify a unique **solicitor** for it.

If \( o \) is a solicitor for resource \( r \), it has the unique responsibility to dispose \( r \).

CLOSER infers a solicitor by:

- First computing a set of **solicitor candidates** from the resource interest graph for each point in the program.
CLOSER proves a resource is unshared if it can identify a unique **solicitor** for it.

If $o$ is a solicitor for resource $r$, it has the unique responsibility to dispose $r$.

CLOSER infers a solicitor by:

- First computing a set of **solicitor candidates** from the resource interest graph for each point in the program

- Then by doing data flow analysis to ensure that the inferred solicitor candidates “agree” at every program point.
Inference of Solicitors

To compute a solicitor candidate for resource $r$:

\[ P = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle \]

CLOSER first computes a set of paths $P$ that reach $r$. It then applies a set of unification rules to determine the existence of a canonical path $l.f_1 \ldots f_n$ that may safely be used to dispose $r$. If such a unique path exists, then $l.f_1 \ldots f_n$ is designated as a solicitor candidate for $r$. If the inferred solicitor candidates for $r$ are consistent, then $r$ is disposed through the cascading series of dispose calls initiated by $l.dispose()$, invoked after the last use point of $l$. 
Inference of Solicitors

To compute a solicitor candidate for resource $r$:

- CLOSER first computes a set of paths
  $\mathcal{P} = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle$ that reach $r$
Inference of Solicitors

To compute a solicitor candidate for resource $r$:

- CLOSER first computes a set of paths
  \[\mathcal{P} = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle\]
  that reach $r$

- It then applies a set of unification rules to determine the existence of a canonical path $l.f_1\ldots f_n$ that may safely be used to dispose $r$
Inference of Solicitors

To compute a solicitor candidate for resource $r$:

- CLOSER first computes a set of paths
  $$\mathcal{P} = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle$$
  that reach $r$

- It then applies a set of unification rules to determine the existence
  of a canonical path $l.f_1\ldots f_n$ that may safely be used to dispose $r$

- If such a unique path exists, then $l.f_1\ldots f_n$ is designated as a
  solicitor candidate for $r$
To compute a solicitor candidate for resource $r$:

- CLOSER first computes a set of paths
  \[ \mathcal{P} = \langle l, f_1 \circ \ldots \circ f_n, \text{May/Must} \rangle \]
  that reach $r$

- It then applies a set of unification rules to determine the existence of a canonical path $l.f_1\ldots f_n$ that may safely be used to dispose $r$

- If such a unique path exists, then $l.f_1\ldots f_n$ is designated as a solicitor candidate for $r$

- If the inferred solicitor candidates for $r$ are consistent, then $r$ is disposed through the cascading series of dispose calls initiated by $l.dispose()$, invoked after the last use point of $l$
Solicitor Example

Inferred solicitor for R:

```
toolBar
button
image
toolBar.dispose()
button.dispose()
image.dispose()
```

```
R
```
Solicitor Example

Inferred solicitor for R:

```
toolBar.button
```
Inferred solicitor for R:

```
toolBar.button
```

Image disposed via call chain:
**Solicitor Example**

- Inferred solicitor for R:
  ```java
toolBar.button
```

- Image disposed via call chain:
  ```java
toolBar.dispose()
```
Solicitor Example

- **Inferred solicitor for R:**
  ```java
toolBar.button
  ```
- **Image disposed via call chain:**
  ```java
  toolBar.dispose()
  button.dispose()
  ```
Solicitor Example

▷ Inferred solicitor for R:

\[
\text{toolBar.button}
\]

▷ Image disposed via call chain:

\[
\begin{align*}
\text{toolBar.dispose()} \\
\downarrow \\
\text{button.dispose()} \\
\downarrow \\
\text{image.dispose()}
\end{align*}
\]
Implementation

- Static Analysis:
  - Builds on IBM WALA framework for analysis of Java byte code
Implementation

- **Static Analysis:**
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit
Implementation

- Static Analysis:
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- Dynamic Instrumentation:
  - Does not rely on modifying the JVM
Implementation

- Static Analysis:
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- Dynamic Instrumentation:
  - Does not rely on modifying the JVM
  - A Manager class keeps dynamic interest counts
Implementation

- **Static Analysis:**
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- **Dynamic Instrumentation:**
  - Does not rely on modifying the JVM
  - A Manager class keeps dynamic interest counts
  - The modified source code calls static methods of the Manager
Implementation

- **Static Analysis:**
  - Builds on IBM WALA framework for analysis of Java byte code
  - Source code transformation utilizes Eclipse JDT toolkit

- **Dynamic Instrumentation:**
  - Does not rely on modifying the JVM
  - A `Manager` class keeps dynamic interest counts
  - The modified source code calls static methods of the `Manager`

- CLOSER appears transparent to the programmer
  - The programmer can inspect and understand the code instrumented by CLOSER
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
We applied CLOSER to automate resource management of an SWT Showcase Graphics Application

~ 7500 lines of code

Uses 67 different resources
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
- Uses 67 different resources
- Reasonably complex resource management logic
Case Study

- We applied CLOSER to automate resource management of an SWT Showcase Graphics Application
- ~ 7500 lines of code
- Uses 67 different resources
- Reasonably complex resource management logic
- Manually removed all resource management code
Case Study, Continued

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>
## Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- User annotates only 5 resources.
- CLOSER infers all the remaining 62 resources.
Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to App Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>
Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- Missing dispose call in the original code was a resource leak.
- Programmer forgot to dispose a Transpose (resource in SWT).
More weak dispose calls because CLOSER is path-insensitive.

- Inserts redundant null-checks even though one already exists.
private void paint() {
    if (image == null) {
        if (image != null) {
            image.dispose();
        }
        image = new Image(...);
    }
}
### Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to App Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- No shared resources in the application.
- CLOSER successfully identified all resources as unshared.
### Case Study, Continued

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td># Resources</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># Strong Static Dispose</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td># Weak Static Dispose</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td># Dynamic Dispose</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># Number of Resource Bugs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># Lines of Resource Mgmt Code</td>
<td>316</td>
<td>356</td>
</tr>
<tr>
<td>Resource Mgmt Code to Application Size Ratio</td>
<td>4.2%</td>
<td>4.9%</td>
</tr>
</tbody>
</table>

- CLOSER doesn’t cause code bloat or substantial runtime overhead.
- And it is correct by construction.
Related Work

**DeLine, R., and Fahndrich, M.**
Enforcing high-level protocols in low-level software.

**Guyer, S., McKinley, K., and Frampton, D.**
Free-Me: a static analysis for automatic individual object reclamation.

**Heine, D. L., and Lam, M. S.**
A practical flow-sensitive and context-sensitive C and C++ memory leak detector.

**Blanchet, B.**
Escape analysis for object oriented languages. application to Java™.
In *OOPSLA* (Denver, 1998).

**Boehm, H.**
Destructors, finalizers, and synchronization.