Jive Research Overview

Towards Scalable Visualizations

April 07 :: Spring 2010

Demian Lessa <dlessa@buffalo.edu>

1. Motivation

A Closer Look at Sequence Diagrams

- Sequence diagrams capture all calls and returns in a program execution.

A Closer Look at Sequence Diagrams

- Sequence diagrams capture all calls and returns in a program execution.
- By their very nature, OO programs tend to:
  - Disperse functionality across multiple classes and methods.
  - Compose objects to create more complex interactions.
  - Instantiate large number of objects to accomplish a program's tasks.
  - Have long execution histories- even programs of modest size!

A Closer Look at Sequence Diagrams

- Sequence diagrams capture all calls and returns in a program execution.
- By their very nature, OO programs tend to:
  - Disperse functionality across multiple classes and methods.
  - Compose objects to create more complex interactions.
  - Instantiate large number of objects to accomplish a program's tasks.
  - Have long execution histories- even programs of modest size!
- As a result, sequence diagrams tend to:
  - Grow horizontally (more objects means more lifelines).
  - Grow vertically (complex iterations means more method activations).

A Closer Look at Sequence Diagrams

- Sequence diagrams capture all calls and returns in a program execution.
- By their very nature, OO programs tend to:
  - Disperse functionality across multiple classes and methods.
  - Compose objects to create more complex interactions.
  - Instantiate large number of objects to accomplish a program's tasks.
  - Have long execution histories- even programs of modest size!
- As a result, sequence diagrams tend to:
  - Grow horizontally (more objects means more lifelines).
  - Grow vertically (complex iterations means more method activations).
- Bottom line: sequence diagrams quickly become unmanageable.

Scalability of Sequence Diagrams

- How do we tackle such problem?

Scalability of Sequence Diagrams

- How do we tackle such problem?
- Here are some ideas:
  - Support (semi-)automatic and manual folding of the diagram.
  - Filter out predefined sets of calls and returns.
  - Project the sequence diagram along specified lifelines.
  - Combine one or more of the above.

Scalability of Sequence Diagrams

- How do we tackle such problem?
- Here are some ideas:
  - Support (semi-)automatic and manual folding of the diagram.
  - Filter out predefined sets of calls and returns.
  - Project the sequence diagram along specified lifelines.
  - Combine one or more of the above.
- For all of the above, we must define appropriate criteria!

Diagram Folding: Agenda

- Define an adequate data structure to abstract the sequence diagrams.
  - Our key data structure is the **Call Tree**.
  - We maintain one call tree per execution thread.
- Identify useful folding criteria.
  - Along the sequence diagram depth (fold subtrees).
  - Along the sequence diagram breadth (fold adjacent siblings).
- Define the necessary folding operations.

Call Tree

- Directed tree.
- Nodes correspond to method activations and edges to method calls.
- Every node n has an associated tuple $\tau(n) = \langle m, e, c, r \rangle$, where:
  - m is the called method,
  - e is the method's execution environment (e.g., an object or a class),
  - $c \in \mathbb{N}$ is the method's call time, and
  - $r \in \mathbb{N} \cup \{\top\}$ is the method's return time.
- Edge (n_1, n_2) encodes a method call from n_1 (caller) to n_2 (callee).
- Total order '<' on call tree nodes: $n_1 < n_2 \Leftrightarrow \tau(n_1).c < \tau(n_2).c$.
- Observation: one call tree per thread!

Figure: Call Tree Annotated with Call Times

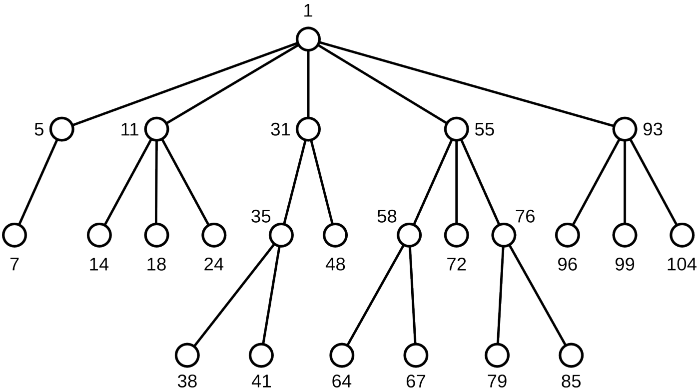
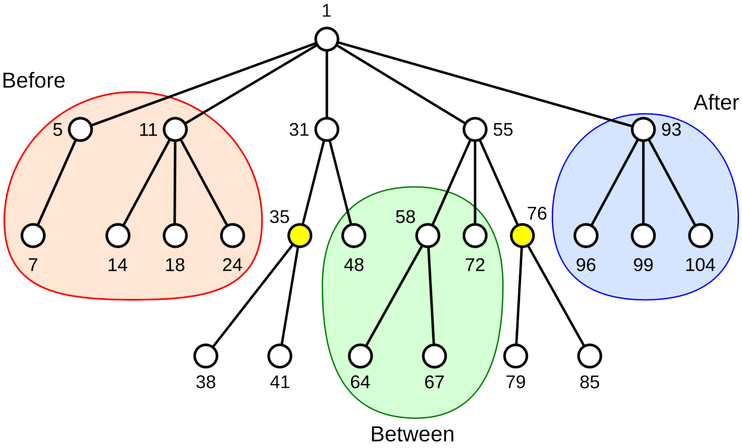


Figure: Possible Uninteresting Regions, Assuming Interest in Activations 35 and 76.



Operations

- `Fold(t,f)`
  - replaces the subtree rooted at f in t with a new leaf node ℓ.
- `FoldBefore(t,n)`
  - folds all nodes f of t such that $f < n$ and $f \notin ancestors(n)$.
- `FoldAfter(t,n)`
  - folds all nodes f of t such that $f > n$ and $f \notin descendants(n)$.
- $\texttt{FoldBetween}(\texttt{t},\texttt{n}_1,\texttt{n}_2)$
  - folds all nodes f of t such that $n_1 > f > n_2$ and $f \notin descendants(n_1) \cup ancestors(n_2)$.
- Note: call trees traversed breadth first.

Figure: Possible Uninteresting Regions, Assuming Interest in Activations 35 and 76 (dup).

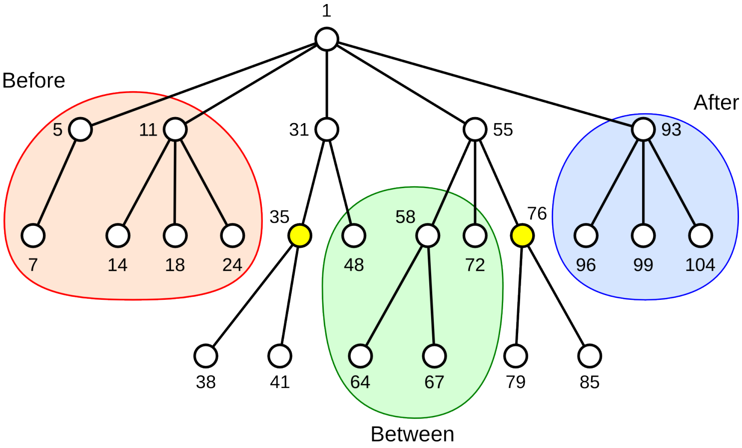
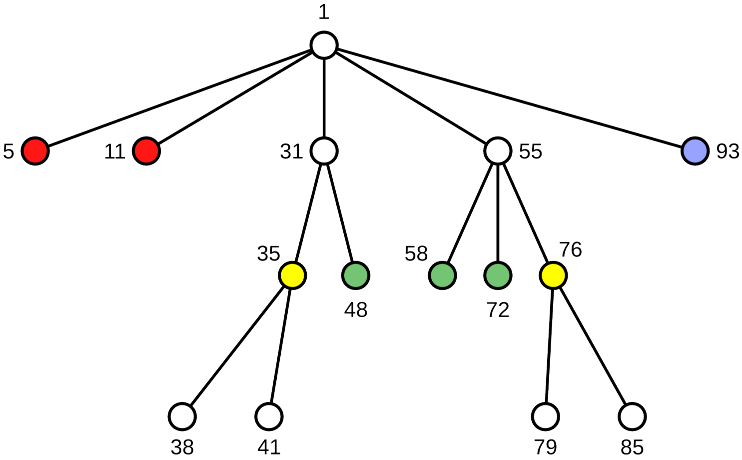


Figure: Call Tree After Folding the Uninteresting Regions.



Operations

- In the last example, nodes 5 and 11 could be further folded into a single node.
- One may argue that this is not necessary.
- Yes, for this example there is no obvious benefit.
- What if the folded call tree had 1000 nodes occurring before node 35?
- The `FoldXXX` procedures presented earlier do not solve this problem.
- We need breadth-wise folding!

Operations (cont.)

- Regular expressions to the rescue!
- Given any reasonably sized sequence of calls made from the same caller, we assume that there are recurring patterns in the call sequence.
- Recurring patterns are usually due to loops, but may occur due to explicit calling patterns in the body of the caller.

Figure: Diagram Folding- Report.java

```
public class Report {

  public String format(String line) { ... }

  public void print(BufferedReader in, PrintStream out) {

    // let N represent the number of lines in the reader
    while (String line = in.readLine() != null) {
      out.printLn(format(line));
    }
  }
}
```

Figure: Diagram Folding- Report.java

```java
public class Report {

  public String format(String line) { ... }

  public void print(BufferedReader in, PrintStream out) {

    // let N represent the number of lines in the reader
    while (String line = in.readLine() != null) {
      out.printLn(format(line));
    }
  }
}
```

Operations (cont.)

- The sequence of calls in `print` may be folded to a single node in the call tree.
- The node is labeled with the regex: $(\texttt{readLine};\texttt{format};\texttt{printLn})^{N}$.
- Such regex compactly represents the call sequence with no loss of information!

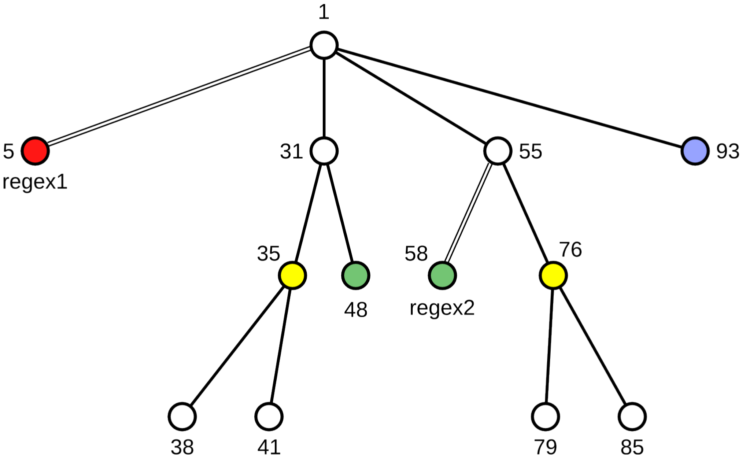
Operations (cont.)

- `RegexFoldBefore(p,n)`
  - computes an ordered list C of all child nodes of p such that $c \in C \Leftrightarrow c < n$;
  - replaces all children of p occurring in C with a single leaf node ℓ labeled with a regular expression computed from the ordered list of method calls obtained from C.
- `RegexFoldAfter(p,n)` and $\texttt{RegexFoldBetween}(\texttt{p},\texttt{n}_1,\texttt{n}_2)$ are defined analogously, with the proper changes to the inclusion criterion of nodes in C.
- Note: algorithm for the conversion of the sequence of method calls into a regular expression is not obvious.

Operations (cont.)

- Let $\texttt{Regex}(\texttt{S})$ be the algorithm that takes a string $\texttt{S}$ as input and returns a regular expression $\texttt{R}$ such that $\texttt{S} \in \texttt{R}$ (i.e., $\texttt{S}$ is a string in the language $\texttt{R}$).
- $\texttt{R}$ is not any regular expression:
  - wildcards and disjunctions are not allowed;
  - only primitive string repeats are allowed: $(ab)^2$ is fine but $(aa)^2$ is not;
  - $\texttt{R}$ is the most compact regex satisfying the above criteria (need not be unique).
- Algorithm:
  - Part I computes all primitive repeats of $\texttt{S}$ in $\texttt{O}(|\texttt{S}| \cdot \texttt{log}|\texttt{S}|)$ time and $\texttt{O}(|\texttt{S}|)$ space;
  - Part II uses Dijkstra's algorithm to compute an optimal regex for $\texttt{S}$ in $\texttt{O}(|\texttt{S}| \cdot \texttt{log}|\texttt{S}|)$ time and space;
  - A simple post-processing step normalizes singletons of $\texttt{R}$ in $\texttt{O}(|\texttt{S}|)$ time and space.

Figure: Call Tree After Regex Folding the Uninteresting Regions.



Filtering

- Filtering is the process by which debug events are omitted from Jive's model.
- As a consequence, call trees no longer contain complete call/return information.
- We introduce **out-of-model** calls and returns to cope with missing information.
- Calls and returns originating and terminating out-of-model are ignored.
- Out-of-model calls to in-model methods are handled as follows:
  - Let n_1 be the largest outstanding in-model node and n_2 the new in-model node.
  - If n_1 has an outstanding out-of-model child o, add n_2 as a child of o;
  - Else, create a new out-of-model node o, add o as a child of n_1, and n_2 as a child of o.
- All other calls and returns are handled as if no filtering was in place.
- Inferring out-of-model calls and returns is done using call trees and call stacks.
- All algorithms introduced thus far may be adapted to handle out-of-model nodes.
- Sequence diagrams must be extended to handle out-of-model calls and returns.

Figure: Extended Sequence Diagram Arrows. Columns indicate whether calls and returns originate/terminate in-model or out-of-model. The ‘in/out+/in’ column indicates a call or return originating in-model and terminating out-of-model, followed by any number of out-of-model activations, terminating in-model after a call or return from out-of-model.

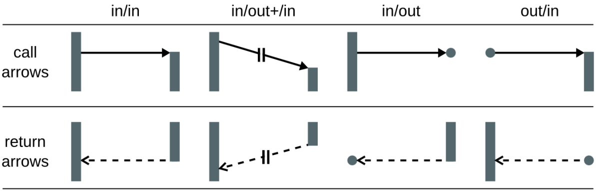
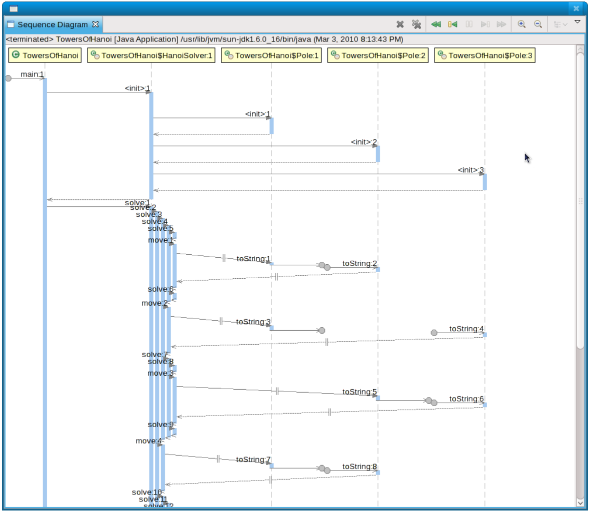


Figure: Extended Sequence Diagram for Hanoi Towers. The towers implement `toString()`, which are called from the out-of-model `PrintWriter.format(String, Object, ...)`.



Filtering (cont.)

- Jive supports all filtering supported by JPDA:
  - class and package filtering based on regular expressions.
- Jive also implements a number of local filters:
  - method and class filtering based on visibility;
  - method filtering based on regular expressions;
  - synthetic method filtering.
- Jive provides default filters for standard Java and Sun packages.

Lifeline Interactions

- Users may be interested in only a subset of lifelines in the sequence diagram.
- Users may project any number of selected lifelines into a new sequence diagram.
- The new diagram only contains method activations occurring in these lifelines.
- Calls and returns from other lifelines are represented as out-of-model.
- Lifelines may be reordered during projection to minimize edge crossings.

Visualizing Query Results

- Query results are typically represented as points in the sequence diagram.
- The `FoldXXX` and `RegexFoldXXX` procedures reduce the diagram's depth and breadth.
- Projecting the lifelines containing result points further reduces the diagram along the relevant objects' lifelines.
- Composing these operations automatically yields the smallest relevant sequence diagram for a given query result set.
- Note: these operations may also be combined manually, by the user, and applied to other use cases.

Status Report

- Currently, Jive supports manual folding of a single node.
- It also fully supports out-of-model calls.
- However, just JPDA filters are currently supported.
- We are in the process of implementing all other algorithms and operations described in here.