State Propagation

Reading assignment

  • Sections 1-4

• For reference only
Data Flow Analysis

- A technique for determining “facts” at locations on a CFG
- Can formulate problems in terms of a data flow framework
  - Know that the problem terminates
  - Know that the computational cost is reasonable
    - Usually $O(n^2)$
  - Can usually define the problem so that the results are conservative
    - E.g., No false negatives
    - Unfortunately, often have false positives

GEN and KILL sets

- For each node $i$, associate sets
  GEN($i$) - what is to be added (generated)
  KILL($i$) - what is to be eliminated (killed)
- The definitions of GEN and KILL depend on the problem that is being solved

- Often the GEN and KILL sets can be derived from the abstract syntax tree
  - E.g., variables defined in a node, DEF($i$)
  - variables referenced in a node, REF($i$)
Data Flow Analysis - General Approach

- Initial node values
  - For each node define GEN, KILL, and OUT
- IN equation
  - For each node we have an equation of the form:
    \[ \text{IN}(n_i) := \text{Merge} \left( \forall n_j \text{ OUT}(n_j) \right) \]
  - “Merge” operation
    - Over the {predecessors|successors} of \( n_i \) depending on whether it is a {forward|backward}-flow problem
    - Merge operator determined by whether it is an all-path or an any-path problem
- OUT equation
  - For each node we have an equation of the form:
    \[ \text{OUT}(n_i) := f_i( \text{IN}(n_i) ) \]
  - Transfer functions \( f_i \) usually depend on GEN and KILL information that is computed for each node
    - Usually: \( \text{OUT} := (\text{IN} \setminus \text{KILL}) \cup \text{GEN} \)

Initial values

- OUT must be defined for all nodes
  - Really, only the nodes that are referenced before OUT is computed
- Initial value depends on problem
  - Often, when the merge operator is intersection, \( \text{OUT}(i) = \text{all possible values} \)
  - Often, when the merge operator is union, \( \text{OUT}(i) = \emptyset \)
DFA Using a Worklist Algorithm

- Define IN value for the initial node and put initial node on the worklist
  - For a forward-flow data-flow problem, the initial node is usually the first node
  - For a backward-flow data-flow problem, the initial node is usually the last node
  - For the initial node, i
    - Often, when the merge operator is intersection, IN(i) = all possible values
    - Often, when the merge operator is union, IN(i) = Ø

- If the worklist is not empty, take a node from the list and process it
  - If the current node gets a new OUT value, add its (successor|predecessor) nodes to the worklist

Fixed point

- If there are only a finite number of possible values that can be associated with a node, and
  - if the function that determines the values that can be associated with a node is monotonic (wrt to a lattice for these values)

- Then the data-flow analysis algorithm will eventually terminate
Compute Final Result

• Usually based on IN, OUT, GEN, and KILL for each node
  • Sometimes need to consider other facts about a node, such as DEF or REF set
  • E.g. reference to an initialized variable
    • Compute reaching definitions
    • Result, for each node, determine if $\text{REF}(i) \subseteq \text{IN}(i)$
  • Sometimes only need to look at the final node

Dataflow Techniques Can Be Used to Detect Anomalous Behavior

• Implicit specification of behavior

  • Use of an uninitialized variable
    Undef $\rightarrow$ Ref, for all or some paths

  • Redefinition of a variable
    Def $\rightarrow$ Def, for all or some paths
Cecil: Olender and Osterweil

• Instead of implicitly defined facts, let the user define application-specific facts (events)

• Prove properties about sequences of events
  • Represented as a Deterministic Finite State Automaton (DFSA) or as a Quantified Regular Expression (QRE)
  • QREs can be translated into DFSAs
  • Non deterministic FSAs can be translated into DFSAs

• Alphabet of the property are the events of concern

Example Property DFSA

Events: open, close, move

• The elevator closes its doors before moving

Assumes the doors are originally closed
Events for Software

- Recognizable events
  - Method calls
    - Can reason about sequences of method calls
    - E.g., Push must be called before Pop
  - Thread interactions
    - Join or Fork
  - Arbitrary operations
    - \(a+b\)
- Need to be able to treat events as indivisible actions
  - E.g., can treat pop and push as atomic as long as they do not contain any events of concern

Annotate control flow graph with events
**All or None Properties**

- An **all** property is a behavior that must always happen on all possible executions
- A **none** property is a behavior that must never happen

Why not a **some** property?

**Example Property DFSA**

**Events:** open, close, move

- The elevator **always** closes its doors before moving
State Propagation

- States of the property are propagated through the CFG
  - For each node of the CFG, indicate the states of the property that the program could be in at that point in the program
  - An any path problem
- An all (none) property is proved if only accepting (non-accepting) states are contained in the final node of the CFG

Annotate control flow graph with events
Verifying Properties Using Data Flow Analysis

- Forward Flow, Any Path Problem
- IN and OUT are sets of FSA states
- \( \text{IN}(n) = \bigcup m \in \text{pred}(n) \text{OUT}(m) \)
- \( \text{OUT}(n) = \bigcup s \in \text{IN}(n) \delta(s, n) \)
  - \( \delta \) is the FSA transition function
- Need to initialize \( \text{OUT}(i) = \emptyset \), for all \( I \)
- For a workist algorithm, need to initialize \( \text{IN}(1) = \text{start node} \)
- Result: Let \( f \) be the final node of the TFG
  - All property: Want \( \text{OUT}(f) \subseteq \text{Accept}(P) \)
  - None property: Want \( \text{OUT}(f) \cap \text{Accept}(P) = \emptyset \)
  - Where \( \text{Accept}(P) \) is the set of accepting states of a property, \( P \)

Elevator Controller

```c
void main()
{
    ... 
    1: if (elevatorStopped)
    { ... 
    3:    openDoors();
    }
    ... 
    5: if (elevatorStopped)
    { ... 
    7:    closeDoors();
    }
    9: moveToNextFloor();
}
```
Elevator Controller

void main()
{
  ... 
  1: if (elevatorStopped)
  {...
  3: openDoors();
  }
  ... 
  5: if (elevatorStopped)
  {... 
  7: closeDoors();
  }
  9: moveToNextFloor();
  }

Property

Elevator Controller
Using Quantified Regular Expressions

- Alphabet, quantification, regular expression

For the events \{open, close, move\} show that for all paths

\(((close \lor move)^*, (open^+, close)^*)^* \text{ open}^*\)

State propagation

- Data Flow Analysis Problem –>
  - lattice \((\hat{F}(S), \subset, \cup)\)
  - function space
  - \(\delta: \hat{F}(S) \rightarrow \hat{F}(S)\)
  - facts at nodes are elements of \(\hat{F}(S)\)
- Propagate states until convergence and check if terminal node in an accepting/non-accepting state of DFSA
Example Meet Semilattice

values = \mathcal{P}((\text{open}, \text{close}, \text{move}))

\bot = \{ \}

T = \{\text{open}, \text{close}, \text{move}\}

Ordering = \subset

Meet = \cup

\begin{itemize}
  \item The elevator never moves while the doors are open
  \item This DFSA is a none property
\end{itemize}
Verifying Properties Using Data Flow Analysis

- Forward Flow, Any Path Problem
- IN and OUT are sets of DFSA states
- \( \text{IN}(n) = \bigcup_{m \in \text{pred}(n)} \text{OUT}(m) \)
- \( \text{OUT}(n) = \bigcup_{s \in \text{IN}(n)} \delta(s, n) \)
  - \( \delta \) is the FSA transition function
- Result: Let \( f \) be the final node of the TFG
  - All property: Want \( \text{OUT}(f) \subseteq \text{Accept}(P) \)
  - None property: Want \( \text{OUT}(f) \cap \text{Accept}(P) = \emptyset \)
  - Where \( \text{Accept}(P) \) is the set of accepting states of a property, \( P \)

Elevator Controller

Worklist: \( 1, 3, 5, 7, 9 \)

Diagram of the Elevator Controller with transitions and actions.
Using Data Flow Analysis to Verify Properties

- Many interesting and important properties can be represented as FSAs or QREs
- Can efficiently determine if the property can be violated
- If it can be violated, can see a trace through the program that causes the violation
  - BUT, may not correspond to an executable path

Desiderata

- Representing properties
- Modeling programs
- Interpreting results
- Any/All interpretation
Verifying method invocations for OO systems

- For OO systems, can reason about legal sequences of method invocations
  - e.g.,
    For events \{pop, push, not_empty\}
    show that for all paths
    \(((\text{push})^*, (\text{not_empty}, (-\text{pop})^*, \text{pop}^*)^*)^*
    (not_empty, (-\text{pop})^*)^*

Overly conservative property, but perhaps desirable for a safety critical system

Ordering Properties

- Can not represent \#Insert > \#Delete
  - N1 deletes, K1 inserts, N2 deletes, k2 inserts, etc.
  - K1 + K2 + ... > N1 + N2 + ...

- But for some N, can show that
  \#inserts ≥ \#deletes
Each delete is immediately preceded by at least one insert

For events \{insert, delete\}
For all paths
\((insert^*, (insert,delete)^*)^*\)

Interpreting Results: When the system cannot verify a property
- Provides a counterexample
- User must decide what is wrong
  - There could be a path through the system that violates the property OR
  - The path that violates the property is not executable OR
  - The property is incorrect
State Propagation: Any/All paths

- All paths or no paths determined by how the last node in the graph is treated, not by using intersection or union

- States of the property are propagated through the CFG
  - For each node of the CFG, indicate the states of the property that the program could be in at that point in the program
    - An any path problem
  - An all (none) property is proved if only accepting (non-accepting) states are contained in the final node of the CFG

Annotate control flow graph with events

If bar is a method call and it contains a foo, then must inline the code for bar

CFG for bar
Inline methods with events

If bar is a method call and it contains a foo, then must inline the code for bar

CFG for bar

1 event/node simplifies analysis

foo, bar → foo

foo → bar

foo → foo
Can reduce the CFG before doing the analysis

Interprocedural Analysis - in-line reduced CFG

If foo2 does not call foo or bar, do not need to inline its code.
Inline methods with events

A summarized representation of bar (and everything that it calls)

Advantages of State Propagation

- Data Flow Analysis determines if the property is valid or not
  - Efficient
  - Always terminates
- Conservative
  - Only validates the property if it is true for all/no possible executions
  - When it can not validate the property, it provides a counter example trace
- Relatively easy to use
  - Do not have to understand how the system works
- Relatively easy to write properties
  - Compared to predicate calculus or temporal logic
**Disadvantages of State Propagation**

- Infeasible paths
- Can not usually express properties about compound data values
  - E.g., For all I, A[I] > A[I+1]
- Can not express general ordering relationships
  - # Insert > # Delete
  - But, can ensure that every delete is preceded by at least one insert
  - Or, can ensure # Insert > # Delete, for some specific n

**Associated support**

- Support for eliminating infeasible paths
  - Some systems apply symbolic execution
  - Sometimes combined with a theorem prover
- Support for visualizing counter example traces
- Support for representing properties
**Summary: Using Data Flow Analysis to Verify Properties**

- Many interesting and important properties can be represented as DFSA or regular expressions
- Can efficiently determine if the property can be violated
- If it can be violated, can see a trace through the program that causes the violation
  - May not correspond to an executable path
  - For OO systems, works well for reasoning about legal sequences of method invocations