Concurrency Analysis Using Reachability Graphs, Temporal Logic and Model Checking

Computer Science 521-621
Fall 2011
Prof. L. J. Osterweil

Material adapted from slides originally prepared by Prof. L. A. Clarke

Static Flow Analysis Architecture

DFA Approach

Model Checking: Overview

• System represented as a (possibly “abstracted”) reachability graph
  – State based=>show the values of all the “relevant” variables
• Properties often expressed in a temporal logic
• Reasoning engine propagates valid subformulas through the graph

Model Checking Approach

Power and Value of These Approaches is Best Seen When Applied to Concurrent Systems

• Concurrent software is much harder, more complicated
• Requires very powerful analysis
• Model checking and DFA shine in this domain
Concurrent systems are complex

- Non-determinism means that
  - the same inputs might produce different outputs on different executions
  - When reasoning about a system there are numerous alternatives to consider
    - Usually more than a human can reasonably consider
- In addition to the problems that can arise with sequential programs, have problems that are unique to concurrent systems
  - Data access problems
  - Synchronization problems

Data Access Anomalies

- Typically you want mutual exclusion
  - shared resource (e.g., data) that should only have a single access at a time
  - e.g., don’t want two travel agents assigning the last seat on a plane
- Typically you don’t want race conditions
  - order of execution affects results
  - undesirable nondeterminism

```plaintext
if initially x = 5, then could output (6, 5), (4, 5), or (5, 5)
```

Interleaved Model of Execution; Examples

```
task A
  x := x + 1;
  write x;
  task A
  x := x + 1;
  write x;

task B
  x := x - 1;
  write x;
  task B
  x := x - 1;
  write x;

If initially x = 5, then
```
```
Infinite wait anomalies (continued)

- Deadlock
  - set of tasks mutually waiting on each other; none can advance
  - Often tasks hold resources that other tasks need
  - Also called, deadly embrace

- Livelock
  - execution does not come to a standstill but none of the tasks can advance

Reachability-based Model Checking: some history

- Originally proposed for hardware
- Early 80’s: E. Clarke and Emerson; Queille and Sifakis
- Late 80’s: Improved algorithms and property notations (E. Clarke, Emerson, Sistla)
- 90’s: Symbolic Model Checking (SMV) and other optimizations (Burch, E. Clarke, Dill, Long, and McMillan)
- Current: Hybrid approaches that combine model checking with
  - Theorem proving techniques
  - Symbolic execution
  - Optimization techniques (e.g., points to analysis)

Reachability Graph

- Models state space
  - Each node represents a possible state in a distributed system
    - States represent the value of all the variables
    - For concurrent systems, this includes the program counter for each task
  - If we only consider the value of the program counter for each task then each state is a vector where the ith element is the current program counter of the ith task
  - \(<pc_1, pc_2, \ldots, pc_n>\)
  - Coarse-grained representation of a RG; doesn’t consider values of variables
  - More powerful analysis requires more complete and detailed state vector
Reachability Graph

- Typically, each edge represents progress in a single task
  - Multiple concurrent events may be possible, but allowing only single events captures all states and simplifies the graph structure (interleaved execution model)
- Only have multiple tasks progress when required by the semantics of the programming construct – E.g., rendezvous
- Only contains states that are potentially reachable from the start state

Reachability Graph Example (clarified)

Use a worklist to build the reachability graph

Useful in determining possible/impossible synchronizations

Identifies two possible synchronizations

But this is “precise up to symbolic execution”
This is a bit harder

Also identifies deadlocks

Analysis of the reachability graph
- State based (what we have just seen)
  - Look at states of individual nodes
    - E.g., deadlock
- (sub)Path based
  - Look at states of sequences of nodes
  - Can do GEN, KILL, IN, OUT, etc.

Comparison of DFA and Model Checking
- Both base analysis of possible “state” at all nodes
- Model Checking usually starts by assuming a more detailed “state” (e.g., values of key variables)
  - Locks for concurrent systems
  - Test predicates for sequential systems
- Propagation can then filter out infeasible paths
- DFA “state” is usually set of states of property automaton
  - Infeasible path filtering added on as needed

Reachability analysis is inherently exponential
- Size of the reachability graph is (worst case) exponential in the number of tasks
  - N nodes per task, T tasks
  - worst case bound on the size of the graph: \( N^T \) nodes in the reachability graph
- Data flow analysis is often quadratic in the size of the graph
  - \((NT)^2\)

Properties in Model Checking
Usually Defined in Temporal Logic
- Considerable expressive power and elegance
  - Can express “live”, “forward dominator”, etc.
- Uses special “temporal” predicates to make it easier to describe “future” states
  - i.e., states that are true about the rest of an execution
- Annotate nodes with specifications of the state of affairs that holds at each node
- Propagate this state information (usually backwards) as temporal predicates

Data flow analysis is often quadratic in the size of the graph
- \((NT)^2\)
**Temporal Logic**

- **Temporal quantifiers**
  - Assume \( p, q \) are atomic propositions
  - \( Gp \) - \( p \) is **globally** true or always true
  - \( Fp \) - \( p \) holds sometime in the **future** or eventually
  - \( Xp \) - \( p \) holds in the **next** state
  - \( p U q \) - \( p \) holds **until** \( q \) holds
  - \( p R q \) - \( q \) is **released** from having to be true if \( p \) is true
- **Path quantifiers**
  - \( A \), \( E \)

**Examples of Temporal Propositions That are True at the Root**

- \( AGp \)
- \( AFp \)
- \( EGp \)
- \( EFp \)

**More Examples**

- \( A(p U q) \)
- \( AXp \)
- \( E(p U q) \)
- \( EXp \)

**More examples**

- \( p R q \) - \( q \) is released from being true if \( p \) is true
  - \( p \) is not required to become true

**Assigning propositions to nodes**

- Mark nodes with propositions that are true at that node
- Each type of expression has a rule for how to propagate that expression through the graph
  - E.g., Mark a node with \( AXp \) (\( EXp \)) if for all (some) of its successors \( p \) is true
Some Examples of More Familiar Propositions

- $p == $ The variable $X$ is used
  - Nice to know $EFGp$ at a node where $P$ is defined
  - Better if $AFp$
- $q == $ Move elevator
  - Nice to know $\neg EGq$ at nodes where elevator door opens
  - Sharper to know that at the root $AG(q \rightarrow s)$ where $s ==$ door is closed

Propagating Propositions: $AXp$ and $EXp$

- $p -$ in the next state, $p$ is true

Propagating Propositions: $AFp$

- $Fp -$ at some time in the future, $p$ is true

Propagating Propositions: $EFp$

- $Fp -$ at some time in the future, $p$ is true

Propagating Propositions

- $Gp -$ globally in the future, $p$ is true
  - “Globally in the future” talks about paths that can be of infinite length
  - Need to identify strongly connected components (SCCs) in the graph
    - A subset of the graph in which every node is reachable from every other node in the subset
    - Can be computed in time linear to the size of the graph [Tarjan 1972]

Propagating Propositions

- $Gp -$ globally in the future, $p$ is true
  - To propagate $AGp$: Identify SCCs
    - If $p$ is true on every node in a SCC, mark every node in that SCC with $AGp$
    - If $p$ is true on a node and all of its successors are marked with $AGp$, mark that node
  - Repeat Step 3 until a fixed point is reached
Propagating Propositions

\[
G \, p - \text{globally in the future, } p \text{ is true}
\]

To propagate EG \( p \):
Mark every node in a cycle where \( p \) is true on every node in that cycle with EG \( p \)
If \( p \) is true on a node and at least one of its successors are marked with EG \( p \), mark that node
Repeat Step 2 until a fixed point is reached

Mutual Exclusion Example

• Need a reachability graph that shows the states (i.e., the values) of the relevant variables to support reasoning about mutual exclusion between two processes
  - process 1 can be null, trying to obtain the lock, or in its critical region \((n_1, t_1, c_1)\)
  - process 2 can be null, trying to obtain the lock, or in its critical region \((n_2, t_2, c_2)\)
  - turn is a variable that indicates which process can obtain the lock \((0,1,2)\)

Example: mutual exclusion protocol [McMillan]

\[
\begin{align*}
\text{(process 1) } & = n_1, t_1, c_1 \\
\text{(process 2) } & = n_2, t_2, c_2 \\
\text{turn} & = 0, 1, 2
\end{align*}
\]

Example Property

• AG\((t_1 \rightarrow AF\, c_1)\)
  - If process 1 tries to get the lock \((t_1)\) then eventually it gets into its critical region \((c_1)\)
• Subformulas
  - AF \( c_1 \)
  - \( t_1 \rightarrow AF\, c_1 \)
  - AG\((t_1 \rightarrow AF\, c_1)\)
• Note, would like to prove this for all processes (e.g. for process 2 as well) but FSV approaches usually must instantiate a fixed configuration of the system (and property)

Example: propagation

\[
AG\((t_1 \rightarrow AF\, c_1)\)
\]

Need to continue propagating

• AG\((t_1 \rightarrow AF\, c_1)\)
• \((t_1 \rightarrow AF\, c_1)\)
  - equivalent to \((\neg t_1 \lor AF\, c_1)\)
Controlling Complexity of Reachability Analysis

- Don’t consider all interleavings of events, only consider “representative” interleavings
  - Valmari, Godefroid, Wolper, McDowell
- Use compositional techniques
  - Analyze reachable states of portions of the model and summarize
- Still have exponential worst-case upper bound

Summary of Reachability Analysis

- Reachability analysis is intuitively appealing, but difficult to implement efficiently (sub-exponentially)
- Techniques exist to control state explosion, but they still carry an exponential upper bound
  - i.e., May be practical on some problems
Some observations: Model Checking

• Worst case bound \( \text{linear} \) in size of the model
  – Model exponential
  – Symbolic model checking encodes Boolean expressions and usually reduces the size of the model
• Experimentally often very effective!
  – Used to verify hardware and designs
  – Trying to develop appropriate abstractions to make it applicable to software systems

A quick look at three approaches to FSV

• Reachability-based Model Checking
• Flow Equations
• Data Flow Analysis
  – FLAVERS

High-Level Architecture of FSV Systems

Some Observations: Data Flow Analysis

• Overall complexity is \( O(N^2S) \)
  – \( N \) is the # nodes in the model
  – \( S \) is the number of states: property x constraints
  – More precisely
    \[
    O(N^2 \cdot S \cdot S \cdot S \cdot \ldots \cdot S \cdot C_1 \cdot \ldots \cdot C_n)
    \]
  – In our experience, many important properties can be proven with a small number of constraints
  • Experimentally: performance sub-cubic
• Usually requires several iterations to determine needed constraints
• Constraints
  – Many automatically generated on request

Data Flow Based Verification: some history

• Mid-70’s: Originally proposed for def-ref anomalies in FORTRAN (Osterweil and Fosdick)
• Early 80’s: Extended to general properties (Olender and Osterweil) & concurrency (Taylor and Osterweil)
• 90’s: Deadlock detection (Mastinola and Ryder); Efficient representation of concurrency & incremental precision improvement (Dwyer and Clarke)
• Recent: Optimizations, Java (Avrunin, Clarke, Cobleigh, Naumovich, and Osterweil)

Data Flow Analysis: FLAVERS

• Flow Analysis for VERification of Systems
• Represents property as a finite-state automaton
• Reasoning engine based on data-flow analysis
• Relatively efficient because of the system model
  – collection of annotated control flow graphs
  – intertask communication and interleavings are represented with additional nodes & edges
  – does not enumerate all reachable states
• over-approximates relevant executable behaviors
  – Uses constraints to selectively improve precision of the model