BloomUnit

Declarative testing for distributed programs
Team and Benefactors

- Peter Alvaro
- Andy Hutchinson
- Neil Conway
- Joseph M. Hellerstein
- William R. Marczak

- National Science Foundation
- Air Force Office of Scientific Research
- Gifts from Microsoft Research and NTT Communications
Distributed Systems and Software

To verify or to test?
Verification

• Formally specify program and behavior
  – e.g., in Promela

• Systematically test inputs and schedules
  – e.g., using SPIN

High investment, high returns
Testing

- Concrete inputs
- Concrete assertions over outputs / state
- No control over asynchrony, scheduling

Pay-as-you-go investment, diminishing returns
Sweet Spot?

- Investment similar to unit tests
- Payoff closer to formal methods
Context

〜 bloom

- Disorderly programming
- Computation as transformation
- Uniform representation of state: collections
Hypothesis

*Database foundations can simplify distributed systems programming*

Successes to date:

1. **Compact, declarative protocol implementations**
   [Alvaro et al., NetDB `09, Alvaro et al., Eurosys `10]

2. **Static analyses for distributed consistency**
   [Alvaro et al., CIDR `10]

3. **Software quality assurance? Yes.**
The database view of testing

1. Declarative assertions
   – Correctness specs as *queries*

2. Constraint-guided input generation
   – Synthesize inputs from FDs and FKs

3. Exploration of execution nondeterminism
   – Apply LP-based analyses to reduce state space
Testing a protocol with BloomUnit
An abstract delivery protocol

module DeliveryProtocol
  state do
    interface input, :pipe_in,
      [:dst, :src, :ident] => [:payload]
    interface output, :pipe_sent,
      pipe_in.schema
    interface output, :pipe_out,
      pipe_in.schema
  end
end
A delivery protocol
A delivery protocol – Best effort

sender

pipe_in

pipe_sent

receiver

pipe_out
A delivery protocol – Reliable
A delivery protocol – FIFO
Declarative Assertions

– Specifications: *queries* over execution traces
  • Timestamped log of flow at interfaces
– Specifications encode *invariants*
  • Queries capture incorrect behaviors
Declarative Assertions

module FIFOSpec
  bloom do
    fail <= (pipe_out_log * pipe_out_log).pairs do |p1, p2|
      if p1.src == p2.src and p1.dst == p2.dst and
        p1.ident < p2.ident and p1.time >= p2.time
        "out-of-order delivery: #{p1.inspect} < #{p2.inspect}"
      end
    end
  end
end

create view fail as
  select 'out-of-order delivery: ' + p1 + ' < ' + p2
  from pipe_out_log p1, pipe_out_log p2
  where p1.src = p2.src and p1.dst = p2.dst
  and p1.ident < p2.ident and p1.time >= p2.time
Declarative Assertions

```ruby
module FIFO
  bloom do
    fail <= (pipe_out_log * pipe_out_log).pairs do |p1, p2|
      if p1.src == p2.src and p1.dst == p2.dst and
         p1.ident < p2.ident and p1.time >= p2.time
        "out-of-order delivery: #{p1.inspect} < #{p2.inspect}"
      end
    end
  end
end
```

```
```
delivery order (timestamps) never deviates from sender order (encoded into ident)"
```
Input Generation
Input Generation

Idea:
- User supplies constraints (in FO logic)
- Search for *models* of the given formula
  - Let a SAT solver do the hard work
- Convert models into concrete inputs
  - Ensure that the models are *interestingly* different

Implementation:
- Use the Alloy[Jackson ’06] language & solver
Input Generation

**Exclusion constraints**

What records *cannot* appear in an input instance

\[
\text{all } p_1, p_2 : \text{pipe}_\text{in} \\
\mid (p_1.\text{src} = p_2.\text{src} \text{ and } p_1.\text{ident} = p_2.\text{ident}) \\
\implies p_1.\text{payload} = p_2.\text{payload}
\]

(ident functionally determines payload)
Input Generation

Inclusion constraints:

What records must appear in an input instance

some p1, p2 : pipe_in
| p1 != p2 =>
  (p1.src = p2.src and p1.dst = p2.dst)

(there are at least two messages between two endpoints)
Execution Exploration
Execution Exploration

- All distributed executions are nondeterministic
- Each concrete input $\Rightarrow$ set of executions
- Message timings / orderings may differ
- Too large a space to search exhaustively!
Execution Exploration

Messages: N
Message orderings: N!
Loss scenarios: $2^N$
Execution Exploration

**CALM Theorem**[Hellerstein ‘10, Ameloot ‘11]:

*Consistency as logical monotonicity*

- Monotonic logic (e.g. select, project, join) is order-insensitive
- Monotonic => race-free
Execution Exploration: Monotonic program

Message orderings: 1
Execution Exploration: Hybrid program

Messages: K
Message orderings: 1
Execution Exploration

Only explore messages orderings when downstream logic is nonmonotonic

\textit{Search only \textasciitilde \textquotedblleft interesting	extquotedblright \ orderings}
Do you need to learn Bloom?

• Yes.
  – But you can take advantage of these techniques without adopting the language

• Requirements:
  – A high-level query language
  – Monotonicity analysis capabilities
  • Prove (or assert) that program fragments are order-insensitive
Queries?
The fold
module FifoPerSource

  state do
    scratch :enqueue_src, [:source, :ident] => [:payload]
    scratch :dequeue_src, [:source] => [:reqid]
    scratch :dequeue_resp_src, [:reqid] => [:source, :ident, :payload]

    table :storage_tab, [:source, :ident] => [:payload]
    scratch :tops, [:source] => [:ident]
  end

bloom :logic do
  storage_tab <= enqueue_src
  tops <= storage_tab.group([:source], min(storage_tab.ident))
end

bloom :actions do
  temp :deq <= (storage_tab * tops * dequeue_src).combos(storage_tab.source => tops.source, storage_tab.ident => tops.ident, tops.source => dequeue_src.source)
  dequeue_resp_src <+ deq do |s, t, d|
    [d.reqid, d.source, s.ident, s.payload]
  end
  storage_tab <= deq { |s, t, d| s }
end
end

module FifoProto

  state do
    interface input, :enqueue, [:source, :ident] => [:payload]
    interface input, :dequeue, [] => [:reqid]
    interface output, :dequeue_resp, [:reqid] => [:source, :ident, :payload]

    scratch :chosen_src, [:ident]
  end
end