Imperial College London

Verlixir: Verification of Message-Passing Systems

Matt Neave

Why must we verify message-passing systems?

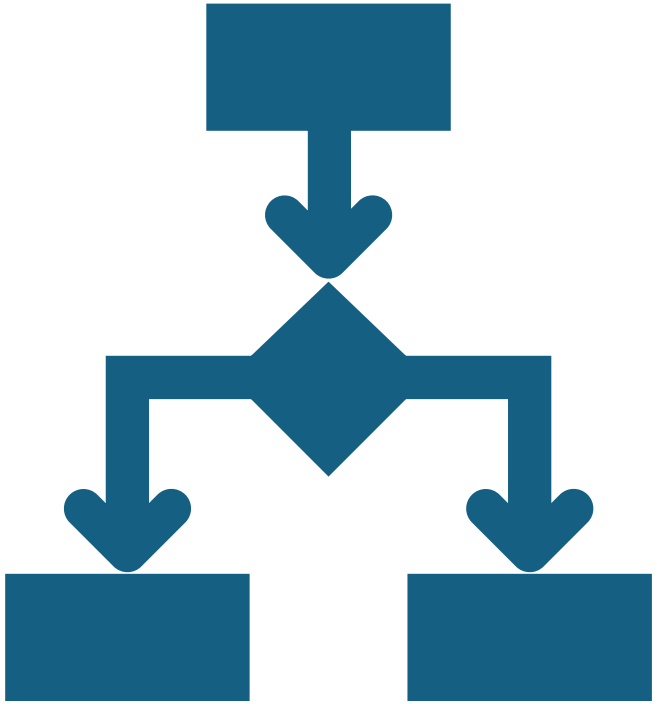
Recent rise in cloud-based clusters

Distributed systems are hard to reason about

Message-passing systems are a common implementation of IPC

Lots of research into the verification of sequential / shared-memory solutions

Lacking research into verification tooling for message passing for actor-based systems



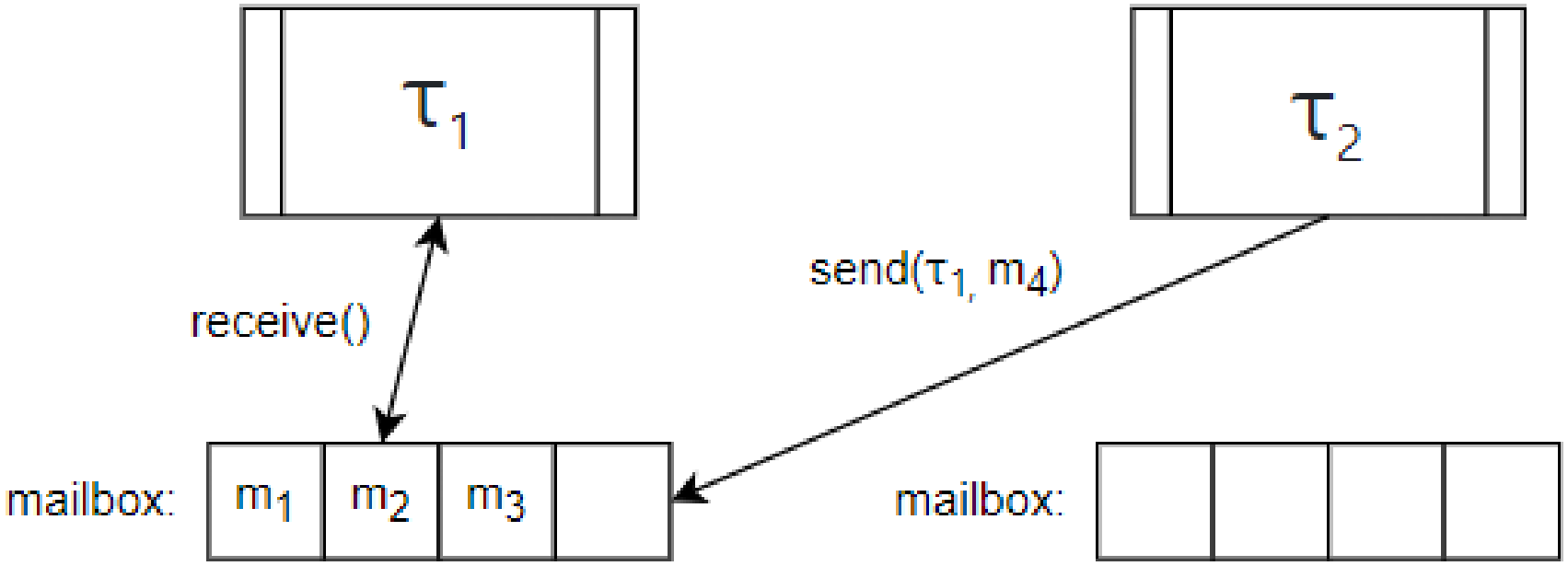
Concurrency

- Multiple computations executed at same time
- Threads / processes communicate with inter-process communication (IPC)
- IPC typically achieved with shared memory or message passing

Actor Model

- Model of computation where actors can:
  - Send messages to other actors
  - Spawn new actors
  - Act in response to a message

Actor Model



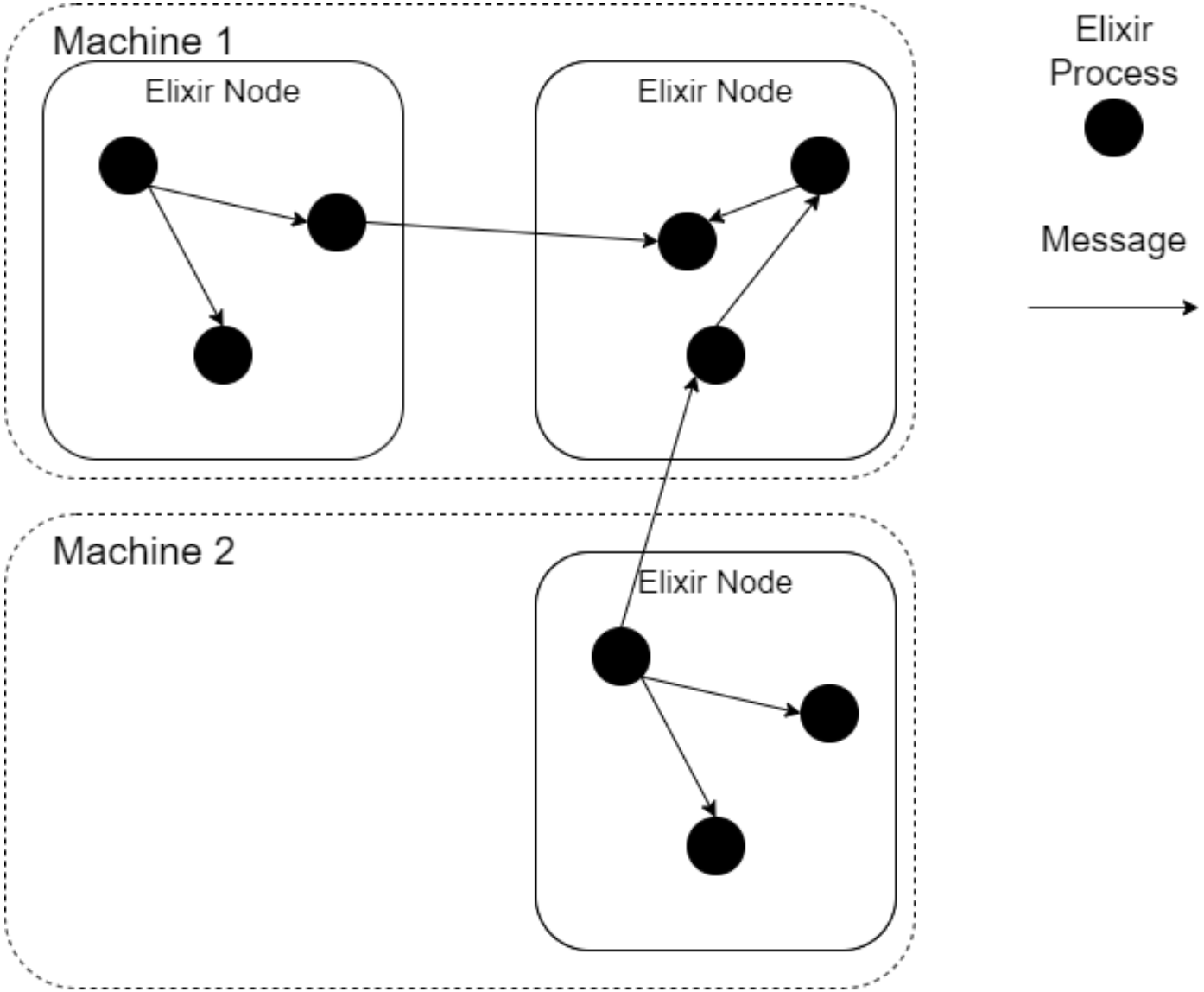
Elixir

- Functional, concurrent, high-level programming language
- Builds on top of Erlang – sharing abstractions for distribution and fault-tolerance

Elixir

```elixir
# Spawn a new process
spawn(fn -> 1 + 2 end)

# Create a new BEAM instance
Node.spawn(:"node1@localhost",
    MyModule, :start, [])
```



Software Verification

Model Checking

Theorem Proving

Model Checking

Determining whether a model of system meets a specification

Typically, exhaustive search of all states (prone to state explosion)

Modern model checkers include **Spin**, PRISM, BLAST…

Verification-aware Languages

- Programming language designed with built-in constructs to facilitate the verification of the program's correctness
- For example, Eiffel, **Dafny**

```
method sqrt(x: nat)
   requires x >= 0
{
   …
}
```

Why must we verify message-passing systems?

Recent rise in cloud-based clusters

Distributed systems are hard to reason about

Message-passing systems are a common implementation of IPC

Lots of research into the verification of sequential / shared-memory solutions

Lacking research into verification tooling for message passing for actor-based systems

Verlixir

- Verification-aware programming language for message-passing systems
- Supports the simulation and verification of systems in a safe, controlled environment
- Provides a specification language with support for linear temporal formulae, predicates and design-by-contract-style functions
- Allows concurrent properties to be parameterized and verified over multiple configurations

Verlixir Simulator

Verlixir Simulator

```
defmodule Coordinator do

  @init
  @spec start_coordinator() :: :ok
  def start_coordinator do
    client_count = 100
    server_count = 10
    ... spawn and link servers / clients ...
  end
end
```

Entry point for Verlixir

Verlixir Simulator

./verlixir –s distributed_braodcast.ex

Starting global broadcast...

South-America broadcasting...

EU-South broadcasting...

...

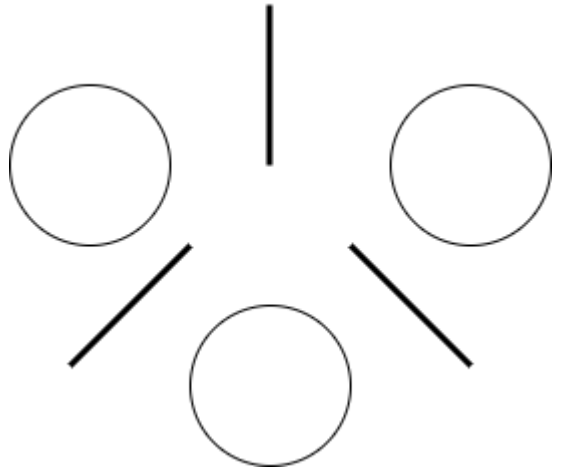
All client acknowledgements received

Because of delays, real-world system may not observe this interleaving for hours / days / weeks...

Verlixir Verifier - Making Elixir Verification-aware

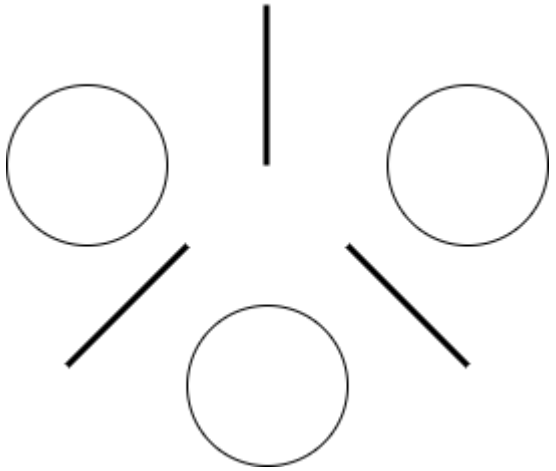
- Deadlock and livelock detection, for FREE
- Linear temporal logic (LTL) to reason about time and change
- Design-by-Contract functions using pre- and post-conditions
- Predicates

Verlixir Verifier: Deadlocks



- Dining philosophers – classical deadlock problem
- Philosophers need two forks to eat!

Verlixir Verifier: Deadlocks



- Dining philosophers – classical deadlock problem
- Philosophers need two forks to eat!

Some simulations output:	Others output:
Picking up 1 Picking up 2 Picking up 2 Picking up 1 All philosophers have finished eating!	Picking up 1 Picking up 2 Picking up 1 Picking up 2 timeout

Verlixir Verifier: Counterexamples

- Verlixir verification mode produces counterexample!

verlixir –v dining_philosophers.ex

Model ran successfully. 1 error(s) found.
The program likely reached a deadlock. Generating trace.

(proc_1 - fork_loop) recv [PICKUP, phil1]
(proc_1 – fork_loop) send [ACK, phil1]
(proc_2 – fork_loop) recv [PICKUP, phil2]
(proc_2 – fork_loop) send [ACK, phil2]

(proc_2 – fork_loop) recv [PICKUP, phil1]
(proc_1 – fork_loop) recv [PICKUP, phil2]

timeout

Verlixir Verifier: LTL

- Same as deadlock counterexamples, Verlixir will produce counterexamples violating LTL formula

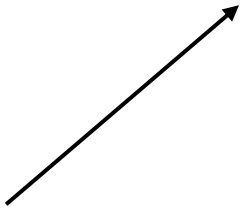
```
defmodule Atm do

  @ltl """
  <>(card_inserted)
  [](card_inserted->(<>[](transaction_failed)||<>[](atm_balance<1000)))
  """
  def start_atm do
    atm_balance = 1000
    card_inserted = false
    transaction_failed = false
    ... implementation ...
  do
end
```

Verlixir Verifier: contracts

- Define pre- and post-condition on contract

```
defv add(x, y), pre: x > 0 && y > 0, post: z == x + y do
  ... defines z somehow ...
end
```



`defv` introduces a contract

Verlixir: predicates

- Simplify LTL claim readability
- Used as expression conditions

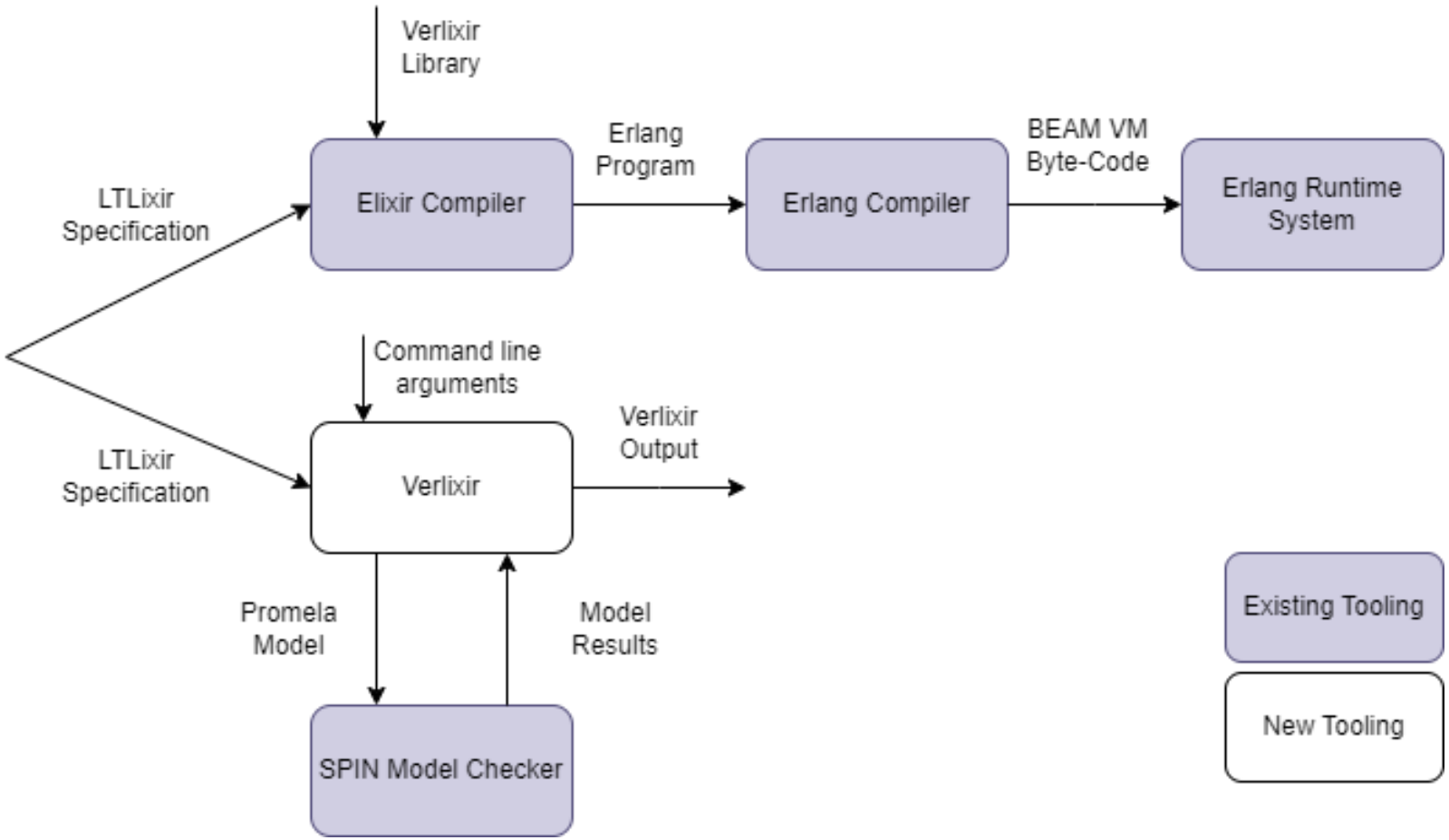
```
@ltl "[](new_round -> <>(consensus)"
def run do
  predicate new_round, messages_received == 0 && messages_sent == 0
  predicate consensus, messages_received >= number_of_nodes / 2 + 1
  ...
  if new_round do
    ...
  end
end
```

Verlixir Verifier: concurrency parameterisation

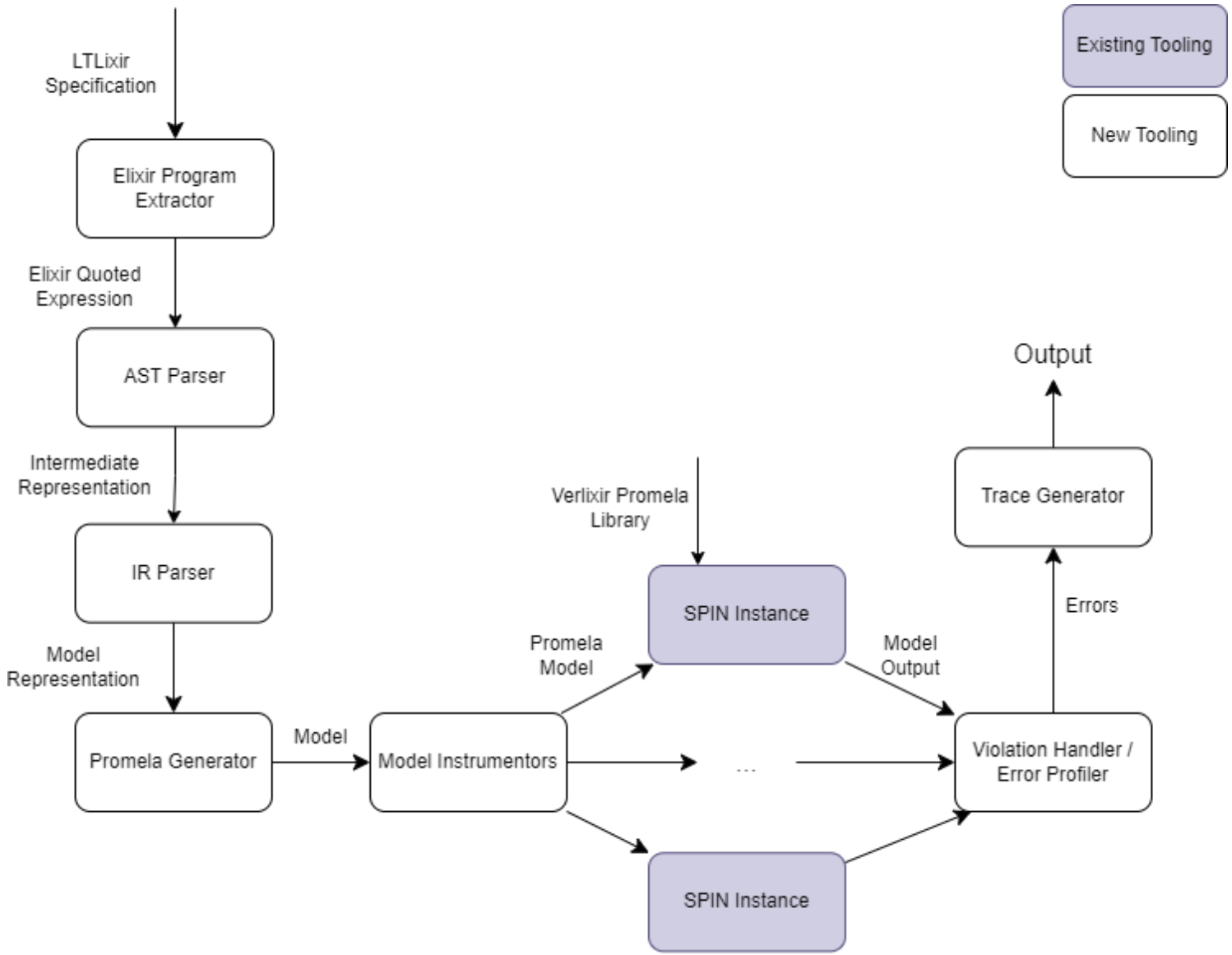
- Model different configurations of variables

```
@model {:client_count, :server_count}
def start_coordinator do
  client_count = 100
  server_count = 10
  ... spawn and link servers / clients ...
end
```

High-level Overview



System Overview



Evaluated algorithms

- Basic paxos
- Consistent hash ring
- Dining philosophers
- Two-phase commit
- Raft (leader election)

Future Work

- Combine model checking with theorem proving to build a more robust verification tool
- Fault injection – determine if systems are fault tolerant by killing processes during verification
- Computation-tree logic (CTL) support, i.e. for algorithms like Raft, where leader election is not guaranteed, we can determine there exists a path where a leader is elected

Final Remarks

- Specifying systems in terms of safety and liveness should be apart of all software development
- Verification-aware languages make system specification more accessible to developers
- With rise of generative AI, we could move towards automatic system implementation from well formed system properties – the use of verification-aware languages can form a feedback loop to ensure the system correctness

Questions?