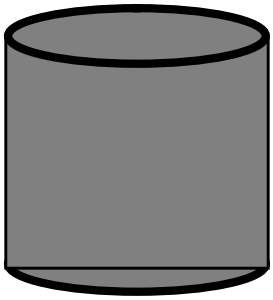
Interfacing System Description Languages to State-Space Exploration-Based Verification

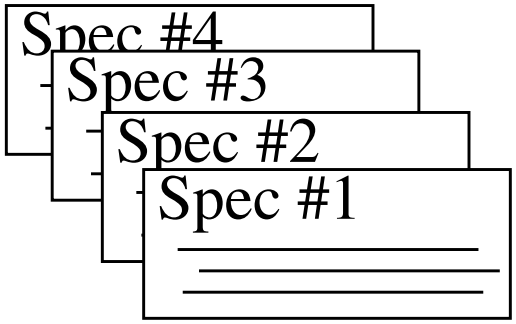
Wendell C. Baker
Electrical Engineering & Computer Science
U.C. Berkeley

October 22, 1993

Design Verification



Design Database



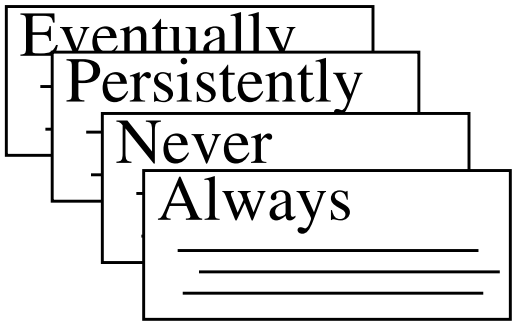
Problem: *Validate the design database against the given specifications*

- Exhaustive simulation is just not possible even for medium sized designs
- Manual selection of test patterns and manual checking of outputs produced
- Simulation time is a major portion of design time

Want: A formal (mathematical) way to prove for all possible inputs for all possible sequences that the specifications are obeyed

Exhaustive State-Space Exploration

System Model



Method:

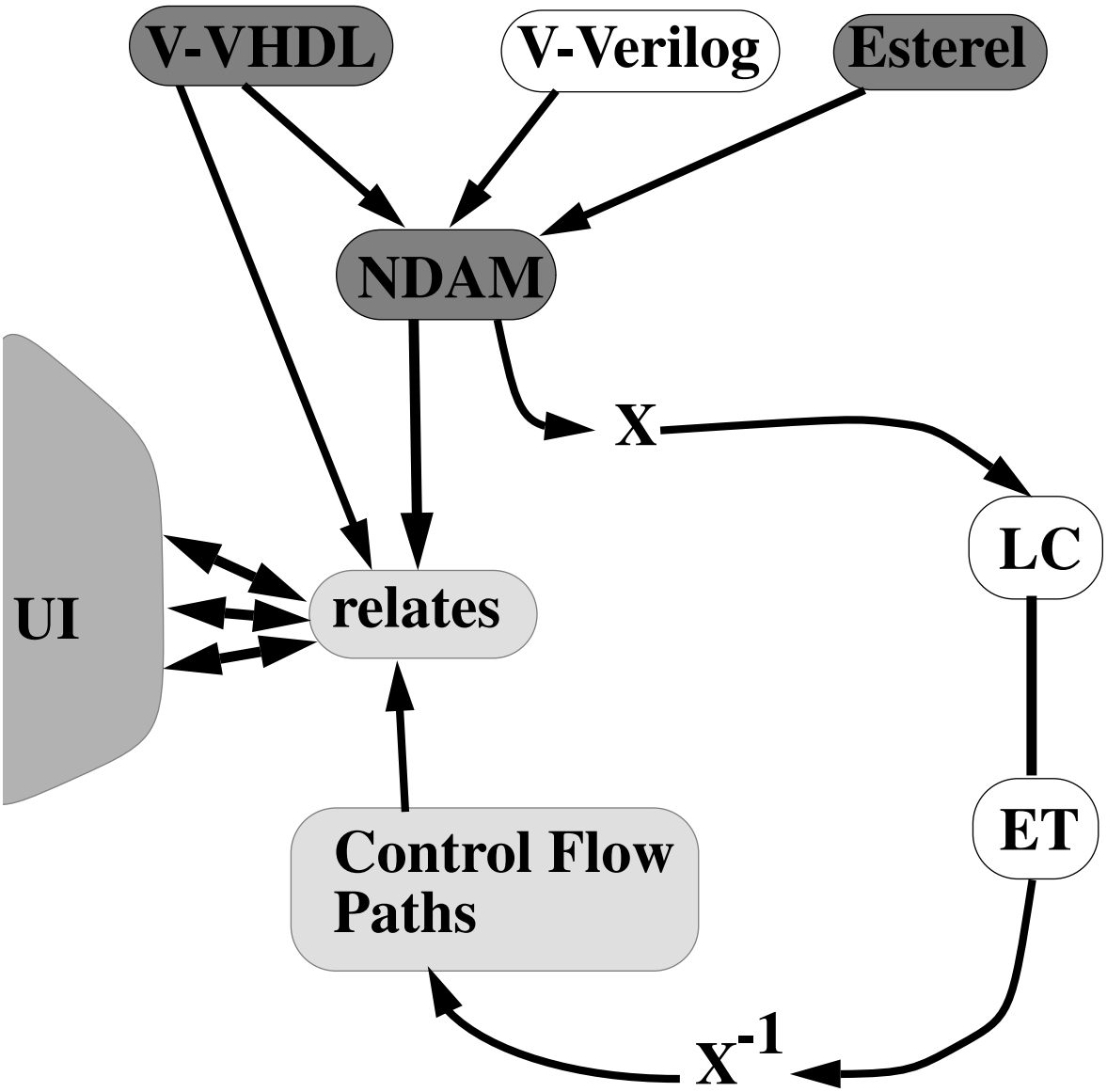
- Examine every state of the system model and determine if any assertion is ever false
- Examine every cycle of the system model and determine if any cycle can fail to perform

Techniques:

Represent the state space and cycles implicitly using Binary Decision Diagrams (BDDs)

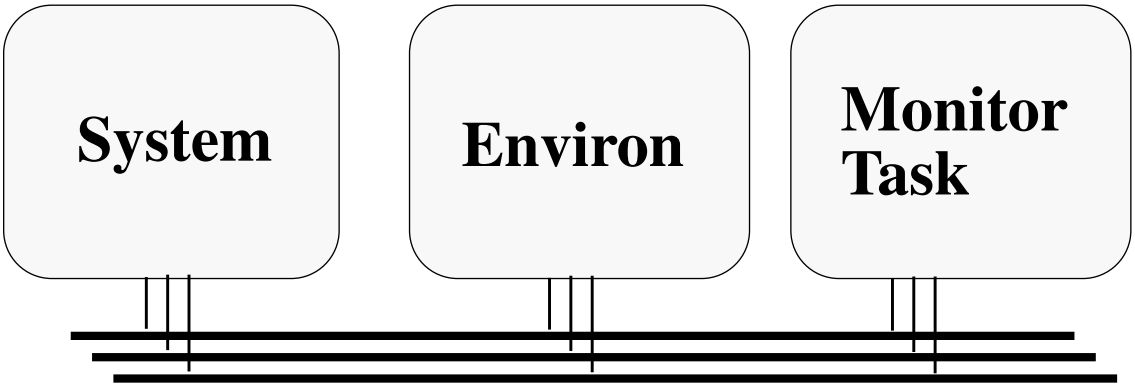
Represent the queries as automata and formulate the problem as one of reached-states or cycle-detection

A Verification-Based Debugger



Language Containment

[Kurshan86]



- The complete system executes under the *Selection/Resolution* model of concurrency
- Language containment asks the question

$$L(System) \subseteq L(Task)$$

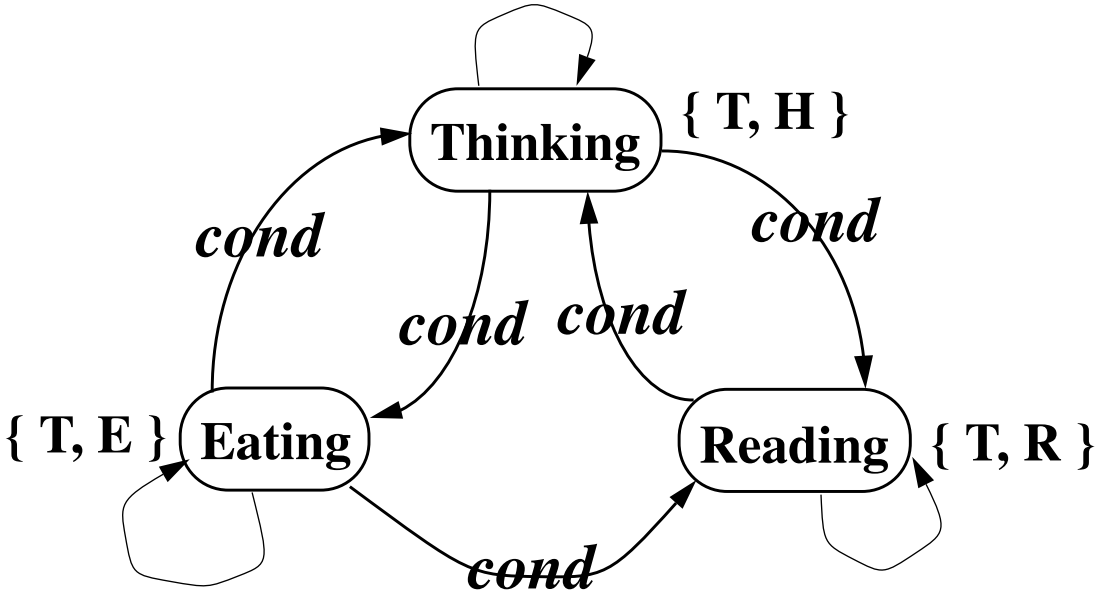
- Can compute language emptiness

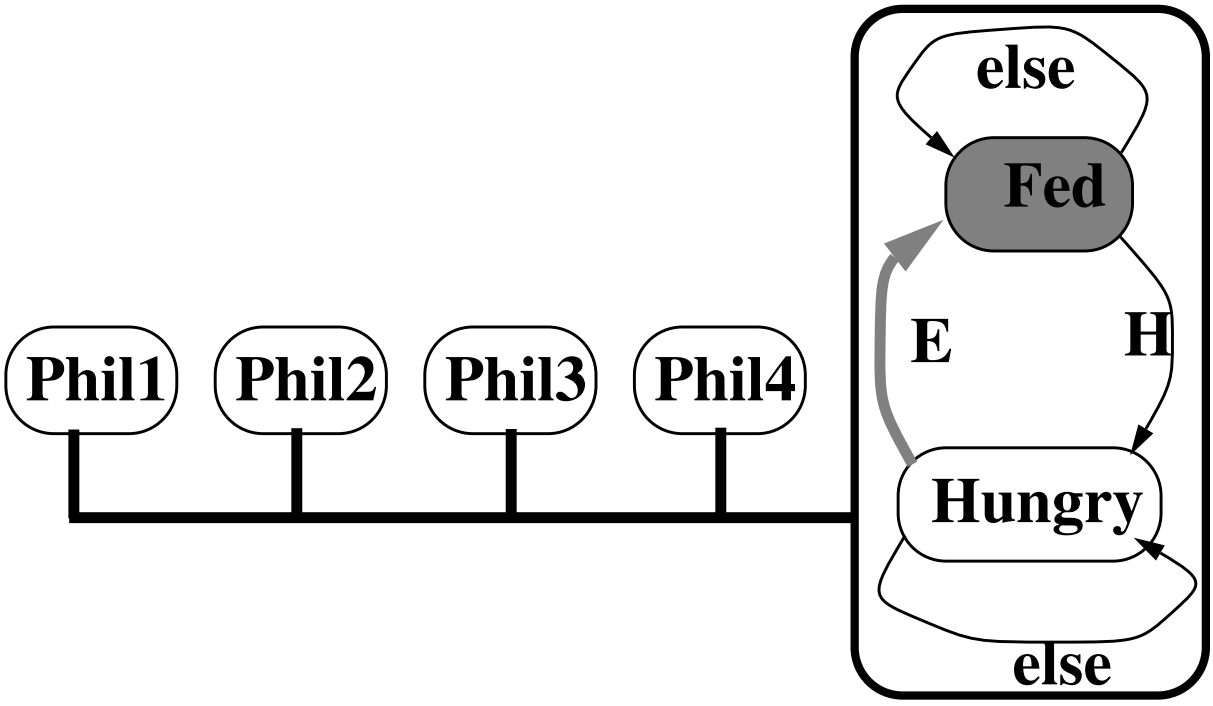
$$L(System) \cap \overline{L(Task)} = \varnothing$$

- If the intersection is non-empty then it represents the set of *error traces* that violate the monitor task

L-process/L-automata

The Dining Philosopher





Three Techniques to Speed Up Query Processing

Map the original description onto a virtual system architecture where the verification algorithms are known to perform well

- Non-Deterministic Abstract Machine (NDAM)
  - causal communication between processes
  - fixed set of transition relations
  - replicated information adds don't cares and allows for multiple good BDD orderings
- Cofactor Tuples

  Tuples of integers can emulate certain well-structured BDDs in the reachable states computation
- Dirty BDD Operations

  Ad hoc pointer manipulations can emulate the next-state computation in memory structures

Outline

- Problem Representation
  - Reactive System Model
  - System and Query Representation

- Basic Algorithms
- Non-Deterministic Abstract Machine
- Modified Algorithms

The Execution Model
A Reactive System

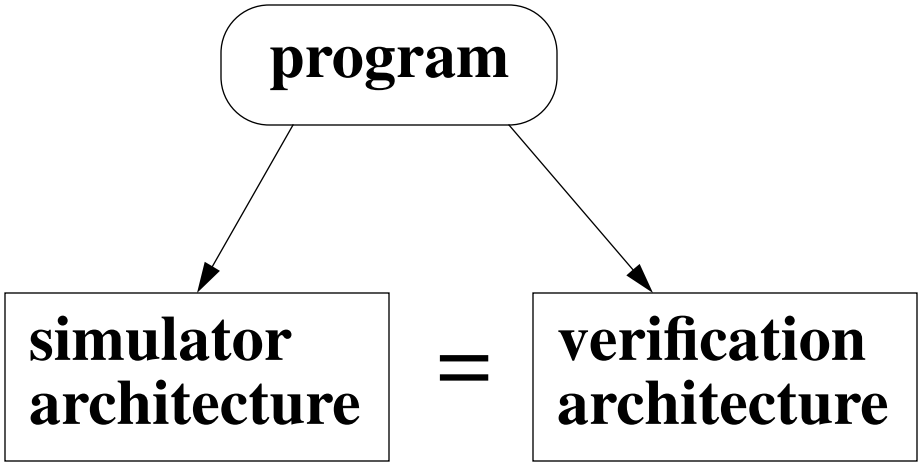
- The system and the environment communicate by means of *events* which *cause* the system to compute a reaction

- Time is abstract -- being represented only in terms of event/reaction pairs

- The system is computes faster than the environment can deliver events -- the *perfect synchrony hypothesis*

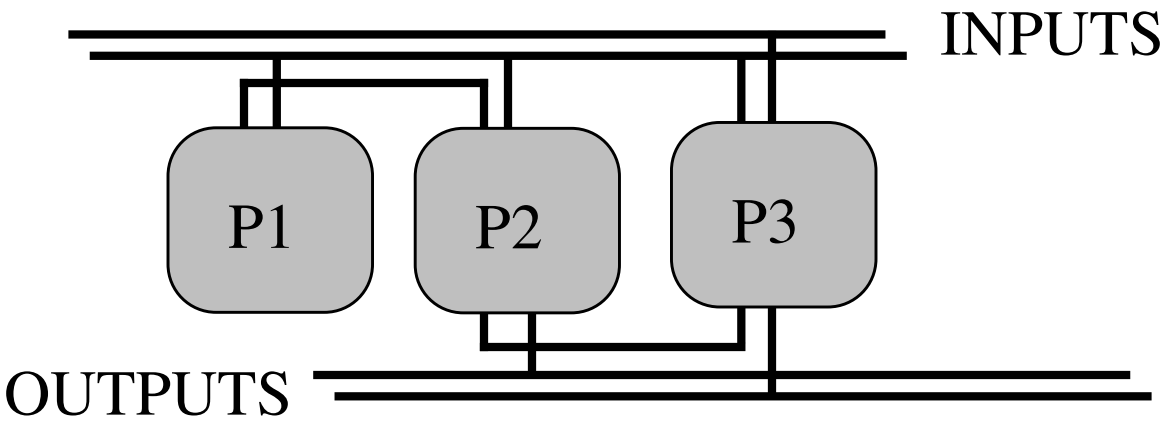
- The system's behavior is defined in only terms of the reactions that it returns

An Automata-Theoretic Semantics for a System Description Language

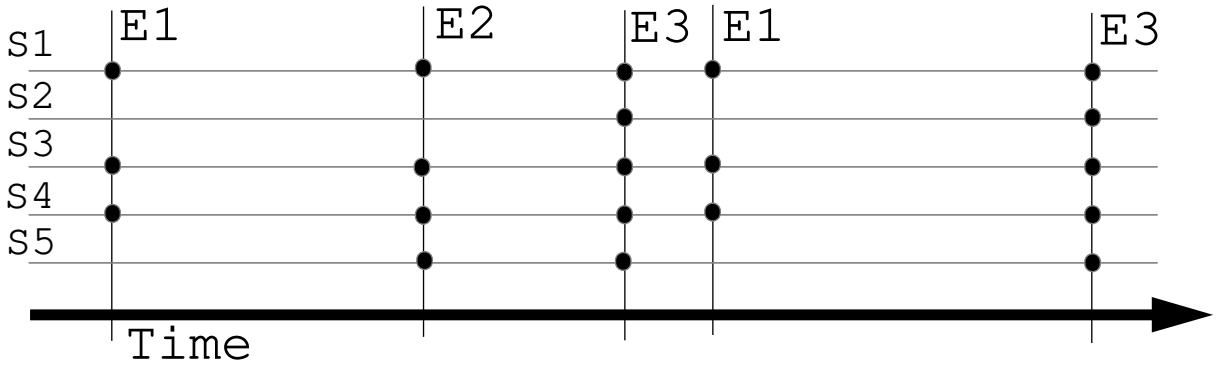


- The simulator architecture defines the meaning of any program in an operational way
- If the simulator is finite-state then there is a way to consider its execution as an automata
- The verification architecture is equivalent and allows for non-deterministic programs as well

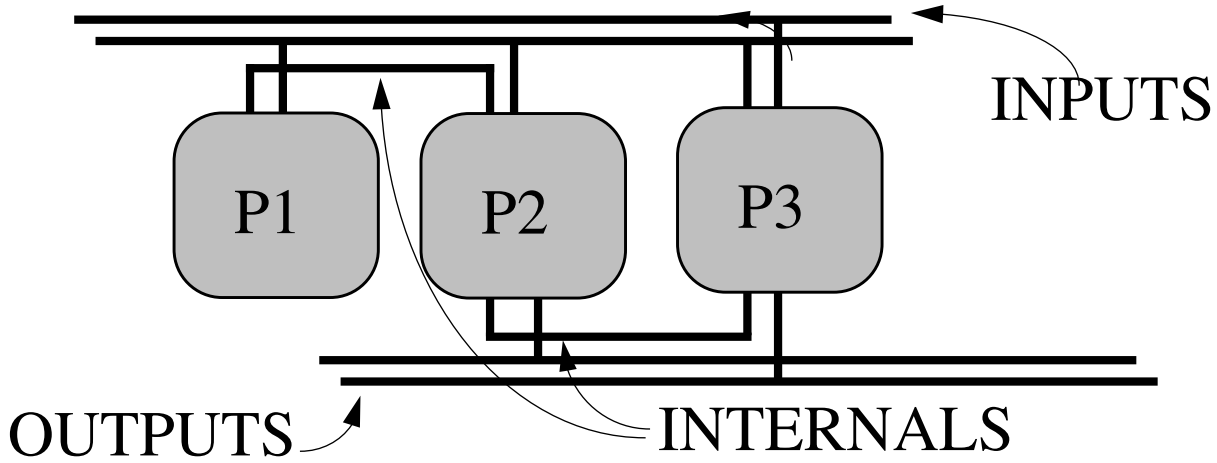
System Description



- A system is as a network of processes
- A process communicates using signals
- Time is measured in terms of events on signals

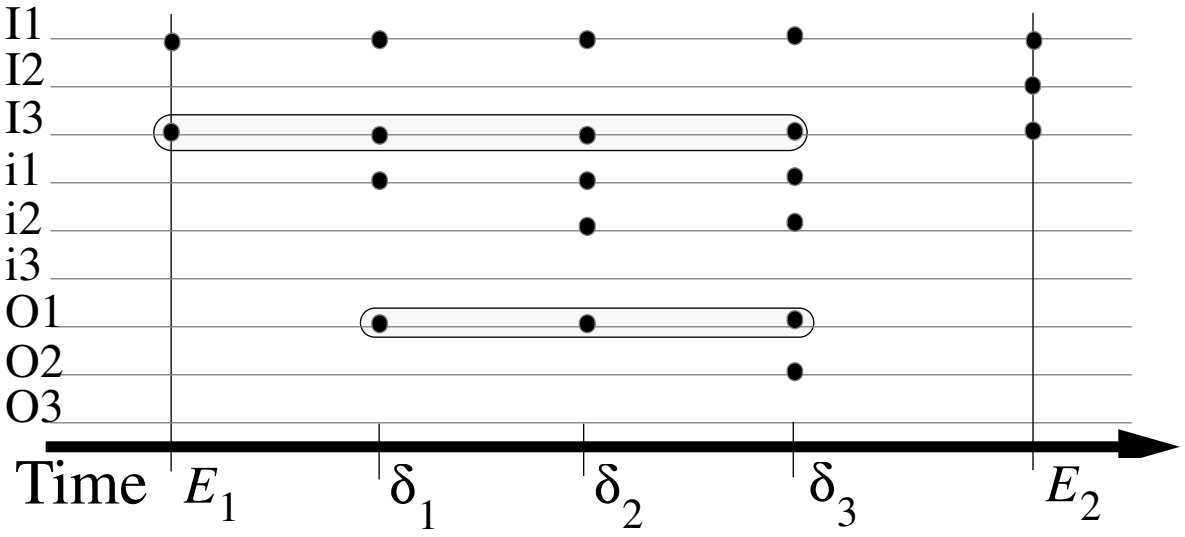


A Causal Reaction



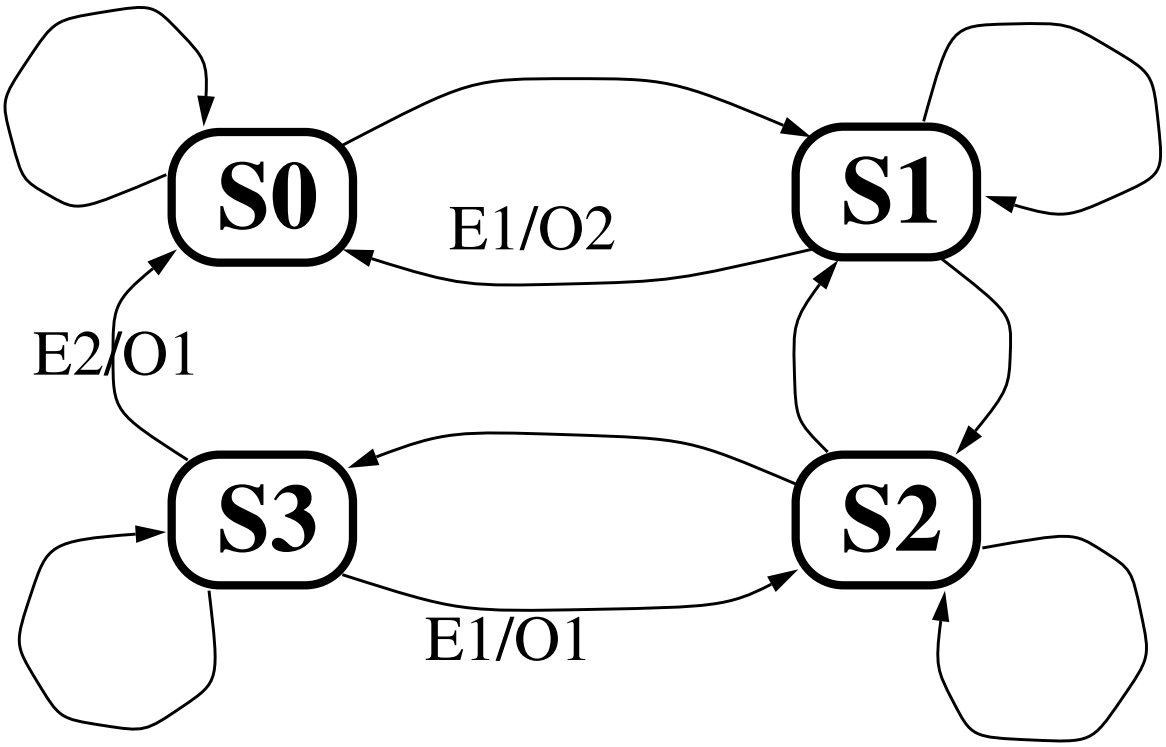
The system's signals consist of *input* signals, *internal* signals and *output* signals

An event consists the presence (absence) of these three components



Causality means that once presence(absence) is determined in an instant, it is never changed

Causal Reactions allow implementations as Finite Automata



Given a state *Si* and an input event *Ij* the reaction computes the output event *Oij*

- table-lookups in software
- finite state machine in hardware

Extensions to the Basic Reactive Process Network

- Subprocesses: want some processes to be activated/deactivated by other processes

  An extra *idling* state to suspend child processes

- Preemption: want parent processes to be able to make other processes go idle or restart

  Consider that the parent can indicate to indicate to a child that it must go idle

- Exceptions: want a child process to transfer control from itself *and* siblings back the parent

  Consider that a child can indicate to the parent and its siblings that the children are going idle and the parent has the control again

Outline

- Problem Representation

- Basic Algorithms
  - Implicit Representations using BDDs
  - **`LC`** - Language Containment

- Non-Deterministic Abstract Machine
- Modified Algorithms

Basic Algorithms and Representations

- **`LC`** - Language Containment

$$\emptyset = \bigcap_i (Cycles(SYSTEM) \cap \overline{CycleSet_i})$$

BDD - Binary Decision Diagram [Bryant86]

- is a canonical form (given an order)
- small for many functions (linear or quadratic)
- represent set, relation, graph
- equality and negation in constant time; other operations can be linear or quadratic

Symbolic Representation

A Fragment of an HDL

```
type bit_vector is
     array(range <>) of bit;
variable a: bit_vector(0 to 1);
variable b: bit_vector(0 to 1);


b(0 to 1):= a(0 to 1);
```

The Transition Relation

$$T(a_0, a_1, b_0, b_1) = \neg(a_0 \oplus b_0) \wedge \neg(a_1 \oplus b_1)$$

Notation

$$T_{SYS}(x, i, y) = \prod T_{p_i}(x, i, y)$$

LC - Language Containment

[Kurshan86][Touati,Brayton,Kurshan91]

1. For each process *P* compute its transition relation

$$T_{P_i}(x, s, r, x) = PT_{P_i}(x, s, r, y) \wedge O_{P_i}(x, s)$$

2. Compute the systemwide transition relation

$$T_{SYS}(x, r, y) = \exists s \left(\prod_i T_{P_i}(x, s, r, y)\right)$$

3. Compute the reachable states of the system

$$R^*(x) = \mu R, R_0(x) \vee (\exists x, R(x) \wedge T_{SYS}(x, r, y)) \; [y/x]$$

4. Restrict the systemwide transition relation to only use the reachable states and non-recur edges

$$T(x, y) = R^*(x) \wedge T_{SYS}(x, r, y) \wedge (r \equiv 1)$$

5. Compute the reachability relation for every state in the transition relation

$$G^*(x, x) = \mu G, T(x, y) \vee \exists i \, (G(x, i) \wedge G(i, y))$$

6. For each cycle set *C* compute the reachable states not covered by it

$$NC_{C_i}(x) = \exists y, G^*(x, y) \wedge G^*(y, x) \wedge \overline{C_i(y)}$$

7. Implicitly enumerate all uncovered cycles

$$\text{success} \Leftrightarrow \varnothing = \bigcap_i NC_{C_i}(x)$$

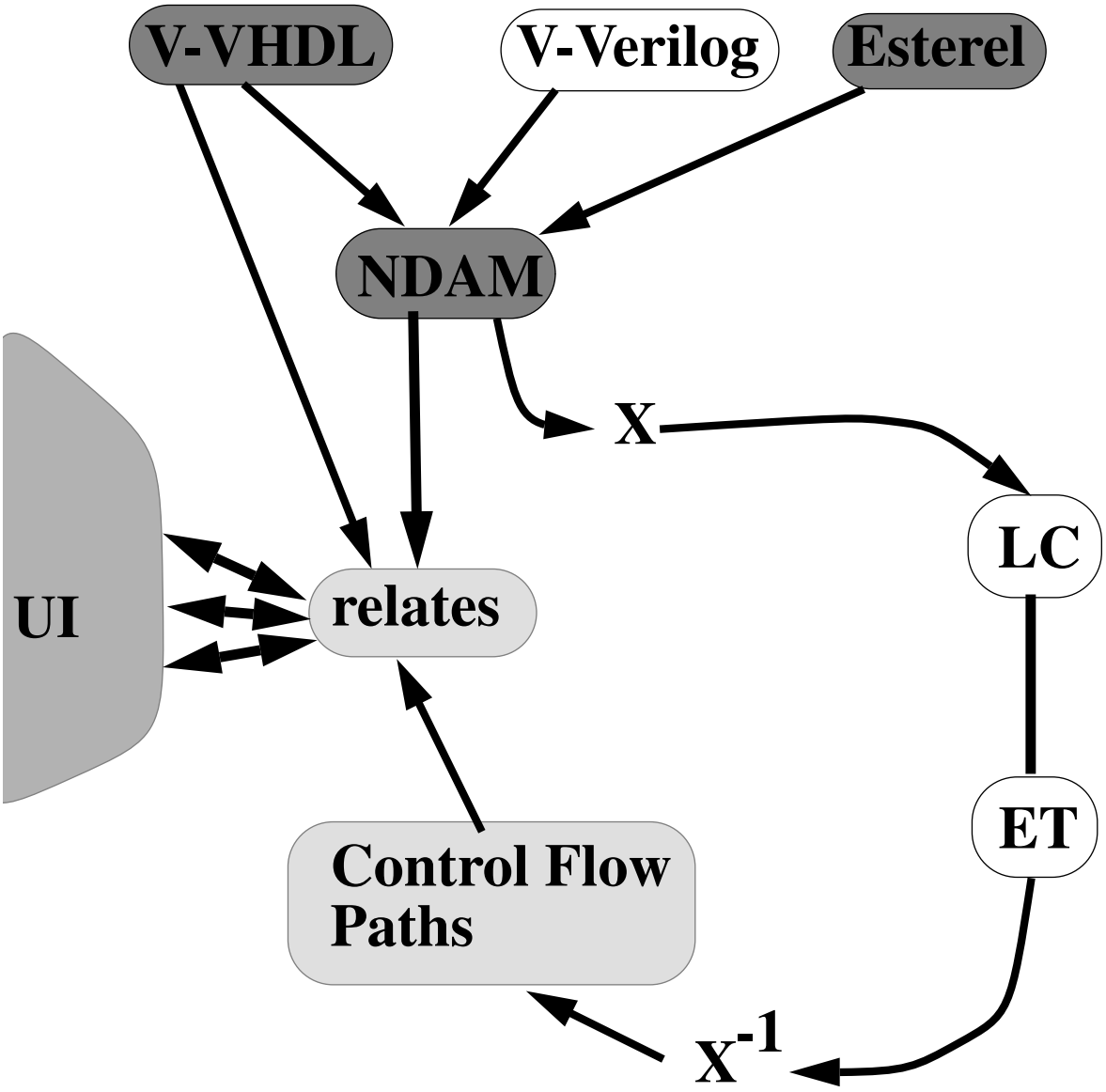
Outline

- Problem Representation
- Basic Algorithms

- Non-Deterministic Abstract Machine
  - Overview of the notation
  - The event-derivative semantics

- Modified Algorithms

An Intermediate Form



- Sufficiently close to the high-level languages
- Can interpret in automata-theoretic terms

Synchronous VHDL

[Baker93]

- Restrict the simulator semantics to allow only for finite-state reactive behavior
- Derive the restrictions on the language from the restrictions on the semantics

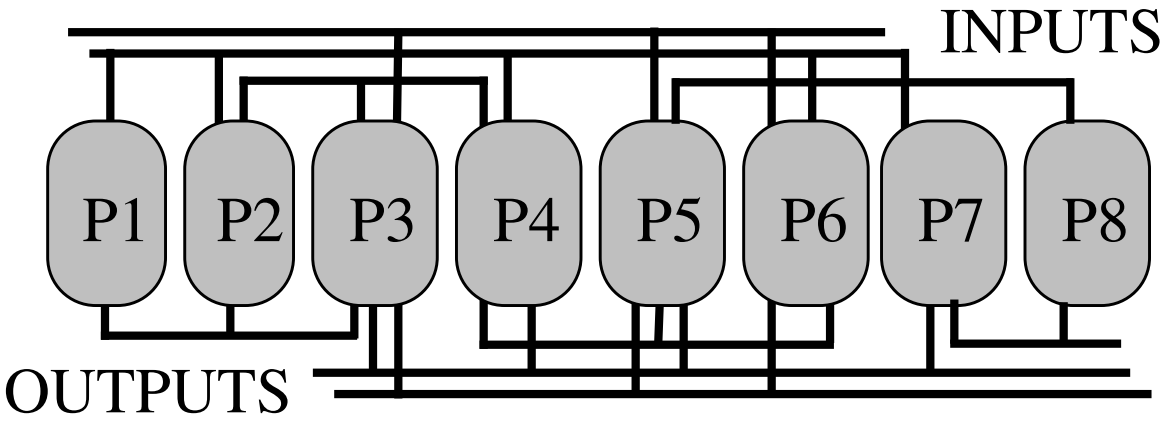
```
signal i, o: bit;

latch:
process
  variable x: bit;
begin
  o <= x xor i;
  x := i;
  wait on i'transaction;
end process latch;
```

- The result is a the largest subset of the VHDL language with finite-state semantics
- Can compile S-VHDL into finite automata

Non-Deterministic Abstract Machine (NDAM)

The network



- The system is a network of finite-state processes *Pi*
- Each process is reactive: occurrence of an event on an input signal *causes* the process to compute its reaction
- There must be a causal execution ordering among the processes in an instant
- A process must compute a non-infinite amount before suspending to await the next event

Non-Deterministic Abstract Machine (NDAM)

Intra-process computation

- Register-transfer level operations on an infinite-register RISC-like machine
- Features:
  - common register-to-register operations
  - signal emission and read
  - goto both deterministic and nondeterministic
  - conditional control flow by if-goto
  - signal event syncronization
  - inter-process control flow
- Control-flow and data-flow are separate and fundamentally different
- Common compiler optimizations and control abstractions can be used

NDAM

Inter-process synchronization

- The execution order of the processes is governed by the *event derivative* of the NDAM network with respect to an input event

$$\frac{\partial Network}{\partial Event} = P_1, P_2, P_1, P_2, P_3, P_4, P_5, P_6, P_3, P_4, P_5$$

- The event derivative is well-defined only when the processes are *causal* according to the rule

  Once presence (absence) of a signal is decided in an instant, the decision cannot be changed

- The event derivative is thus fixed-point of some function describing the intra-instant next-state relations

- Non-causality exists when there is zero or more than one fixed point to this relation

The Event Derivative

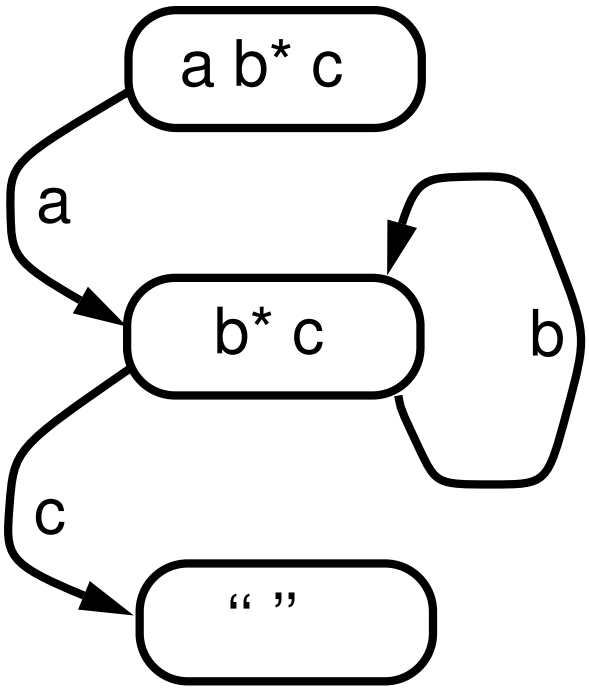
First derived for Regular Expressions
[Brzozowski64]

$$R = a\ b^*\ c$$

$$\frac{\partial R}{\partial a} = b^*\ c$$

$$\frac{\partial R}{\partial b} = b^*\ c$$

$$\frac{\partial R}{\partial c} = \text{“ ”}$$

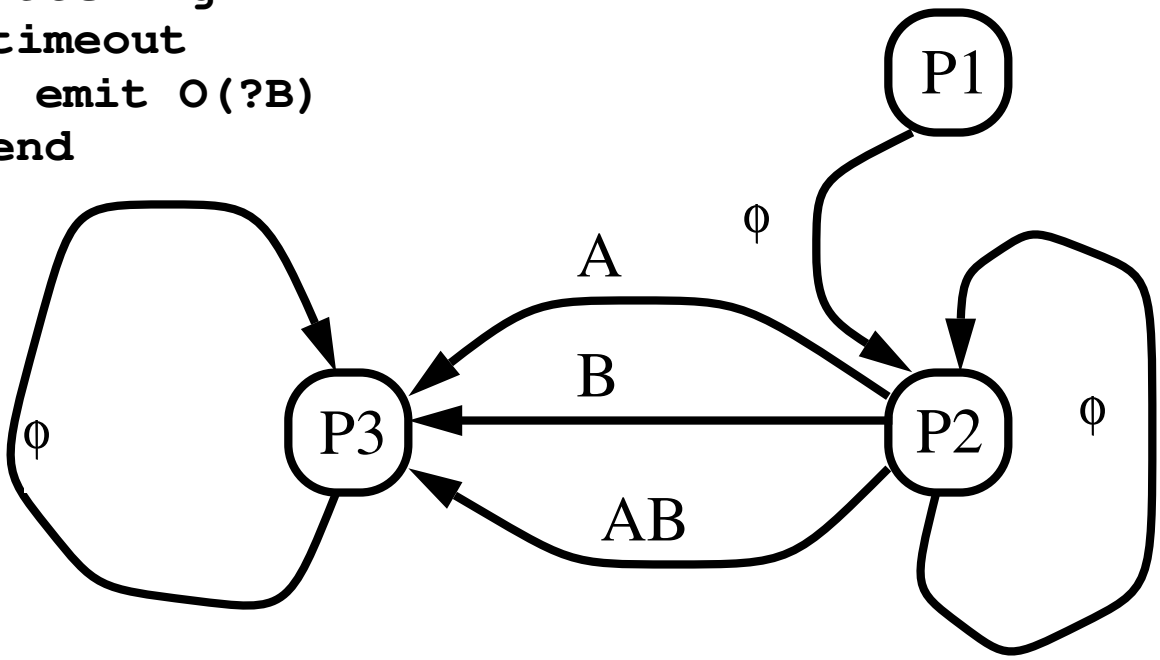


The derivative of *R* with respect to a symbol *c* represents the substring of the pattern possible *after* that symbol

Event-Derivative Semantics

Semantics of Esterel [BerryGonthier88]

```
module M:
input A(integer), B(integer);
output O(integer);
do
  await A;
  emit O(?A)
watching B
timeout
  emit O(?B)
end
```



The derivative of *P* with respect to an event *E* represents the behavior of the program possible *after* that event: the continuation

Event Derivatives for NDAM

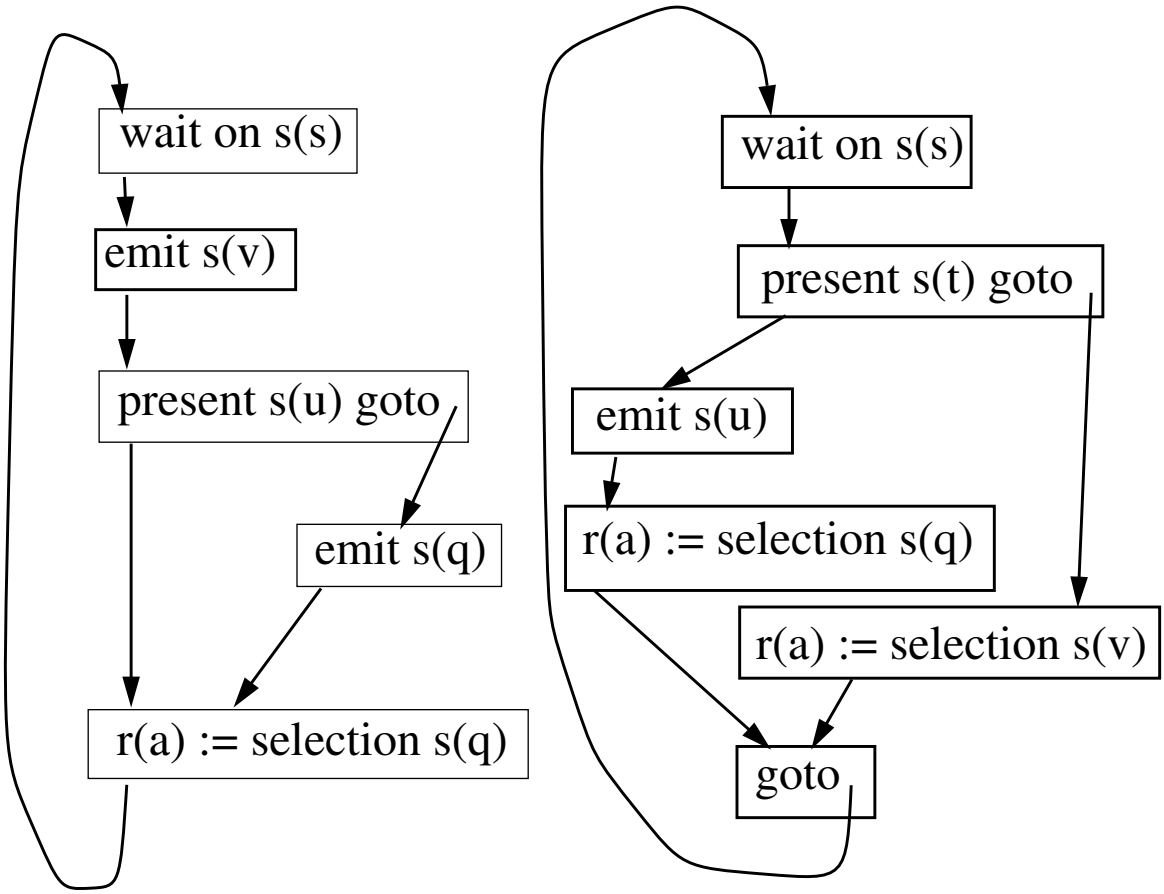
- It is problematic to define an event-derivative operator directly on VHDL networks
- An intermediate level covenient for compilation and event derivative computation

```
type T(bit) 2
signal S(i) T(bit)
signal S(o) T(bit)
register R(x) T(bit)
temporary R(t1) T(bit)
temporary R(t2) T(bit)
L(1):
    R(t1) := selection S(i)
    R(t2) := R(t1) xor R(x)
    emit S(o) R(t2)
    R(x) := R(t2)
    wait on S(i)
    goto L(1)
```

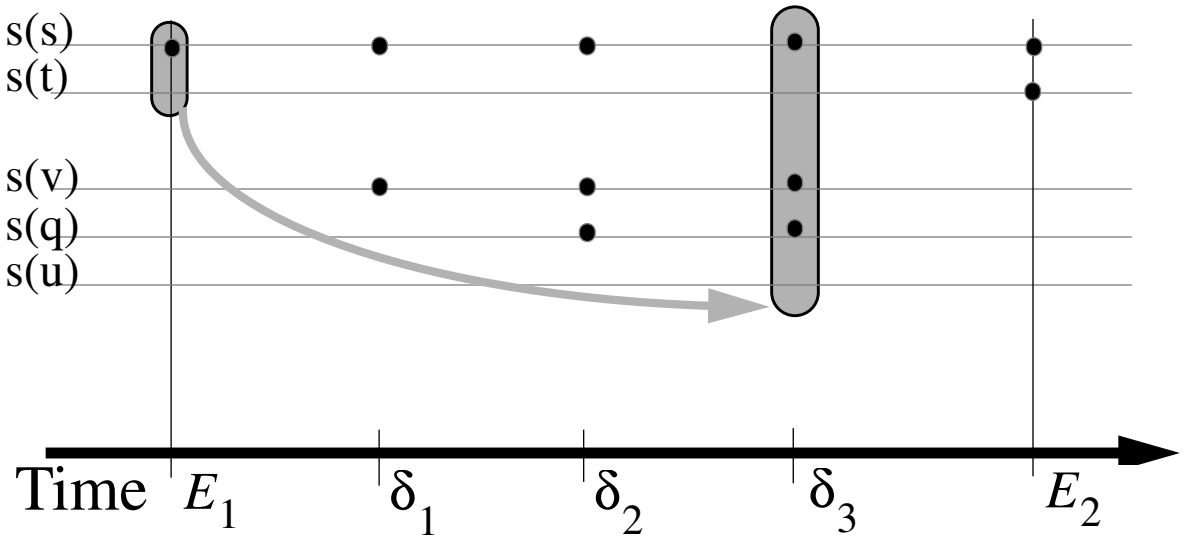
Event Derivative implies Causally-Interleaved Concurrency

$$\frac{\partial Network}{\partial Event} = P_1, P_2, P_1, P_2, P_3, P_4, P_5, P_6, P_3, P_4, P_5$$

To *execute* the network derivative each process executes some instructions until all are stable



Interleaved Concurrency in Causal Delta Time



- The steps of causally-interleaved execution are the delta-time steps

- Each basic block process is equally runnable when its entry information (*EN*) is available

EN = { S(q), S(t) }

r(a) := selection S(q)
present S(t) goto L(9)

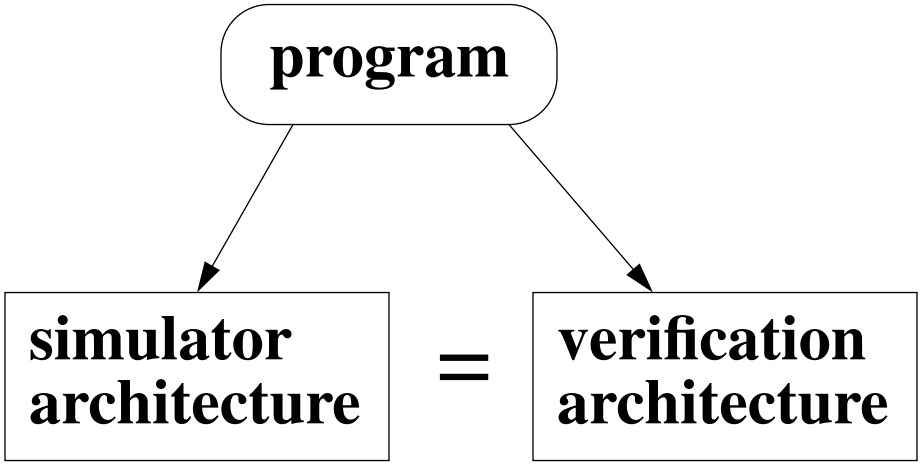
EN = { }

emit S(v) using R(8)

EN = { }

wait on S(r)

Causally-Interleaved Selection/Resolution



- Simulator architecture choices
  - Can compute the EN sets on the fly
  - Can take the network derivatives and implement the automaton
- Verification architecture
  - Can use the synchronous interpretation
  - Can use the interleaving to simpify the transition relation representation

Interleaved Execution

- Synchronous execution implies that every process executes at every cycle.

  The system transition relation is the *conjunction* of the process transition relations:

$$\hat{T}_{SYS}(x, y) = \prod \hat{T}_{P_i}(x, y)$$

- Asynchronous execution implies that each process executes while the other processes must stay stationary.

  The system transition relation is the disjunction of the process transition relations:

$$\hat{T}_{SYS}(x, y) = \sum \hat{T}_{P_i}(x, y)$$

This is important for smoothing distributes over disjunction: *early quantification*

A Fixed Set of Transition Relations

Can exploit optimizations possible on that one next-state expression.

Given a process transition partitioned in two parts:

$$T_{P_i}(x, y) = T_{P_i}^{(op)}(x, y) \wedge T_{P_i}^{(preserve)}(x, y)$$

Then

$$R_{n+1, P_i}(x) =$$
$$(\exists x, R_{n, P_i}(x) \wedge T_{P_i}^{(op)}(x, y) \wedge T_{P_i}^{(preserve)}(x, y)\,)_{[y/x]}$$

However, both of *Top* and *Tpreserve* are rather simple on their own; only their conjunction has a large complicated BDD representation.

Identify ways of speeding up this one next-state computation: *cofactor tuples*

Outline

- Problem Representation
- Basic Algorithms
- Non-Deterministic Abstract Machine

- Modified Algorithms
  - Problems to solve
  - Solutions via Causally-Interleaved Selection/ Resolution

LC - Language Containment

1. For each process *P* compute its transition relation

$$T_{P_i}(x, s, r, x) = PT_{P_i}(x, s, r, y) \wedge O_{P_i}(x, s)$$

2. Compute the systemwide transition relation

$$T_{SYS}(x, r, y) = \exists s \left(\prod_i T_{P_i}(x, s, r, y)\right)$$

3. Compute the reachable states of the system

$$R^*(x) = \mu R, R_0(x) \vee (\exists x, R(x) \wedge T_{SYS}(x, r, y)) [y/x]$$

4. Restrict the systemwide transition relation to only use the reachable states and non-recur edges

$$T(x, y) = R^*(x) \wedge T_{SYS}(x, r, y) \wedge (r \equiv 1)$$

5. Compute the reachability relation for every state in the transition relation

$$G^*(x, x) = \mu G, T(x, y) \vee \exists i (G(x, i) \wedge G(i, y))$$

6. For each cycle set *C* compute the reachable states not covered by it

$$NC_{C_i}(x) = \exists y, G^*(x, y) \wedge G^*(y, x) \wedge \overline{C_i(y)}$$

7. Implicitly enumerate all uncovered cycles

$$\text{success} \Leftrightarrow \varnothing = \bigcap_i NC_{C_i}(x)$$

Problems

2. Computing the system transition relation in direct form is extremely expensive

$$T_{SYS}(x, r, y) = \exists s \left(\prod_i T_{P_i}(x, s, r, y)\right)$$

3. Computing the complete set of set of reachable states can be expensive

$$R^*(x) = \mu R, R_0(x) \vee (\exists x, R(x) \wedge T_{SYS}(x, r, y))\ [y/x]$$

5. Computing the transition relation transitive closure is very expensive

$$G^*(x, x) = \mu G, T(x, y) \vee \exists i\,(G(x, i) \wedge G(i, y))$$

Solutions

2. Use causal interleaving to avoid computing the product machine directly

$$T_{SYS}(x, r, y) = \exists s \left(\sum_i T(x, s, r, y) \right)$$

3. Use algorithms that can be run with only a subset of the complete reachable state set *R*(x)*.

Iterate until you reach all of *R*(x)*
- compute a subset of *R*(x)*
- check for failures

5. Use cycle-detection algorithms that do not need the transitive closure computation

Focus on the sets of *states* that participate in cycles rather than the set of edges that participate in cycles

Summary

- The design process consists of building a software model of the system and *verifying* its behavior through simulation
- State space methods can be used to *formally verify* properties of the system under a concurrency model
- The Synchronous Selection/Resolution (S-S/R) concurrency model forms the basis for using language containment algorithms
- Meaningful subsets of system description languages which are consistent with S-S/R can be identified
- *Causally-Interleaved Selection/Resolution* is an approximation to S-S/R which offers the potential for faster language containment algorithms