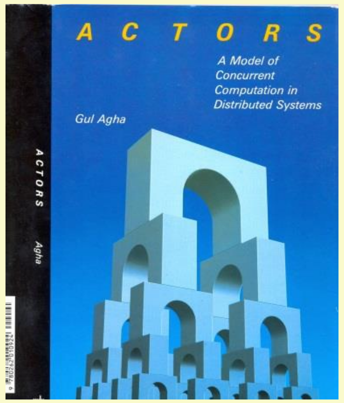
Abstractions, Semantic Models and Analysis Tools for Concurrent Systems: Progress and Open Problems

Gul Agha
University of Illinois at Urbana-Champaign
Embedor Technologies
http://osl.cs.uiuc.edu



Acknowledgements

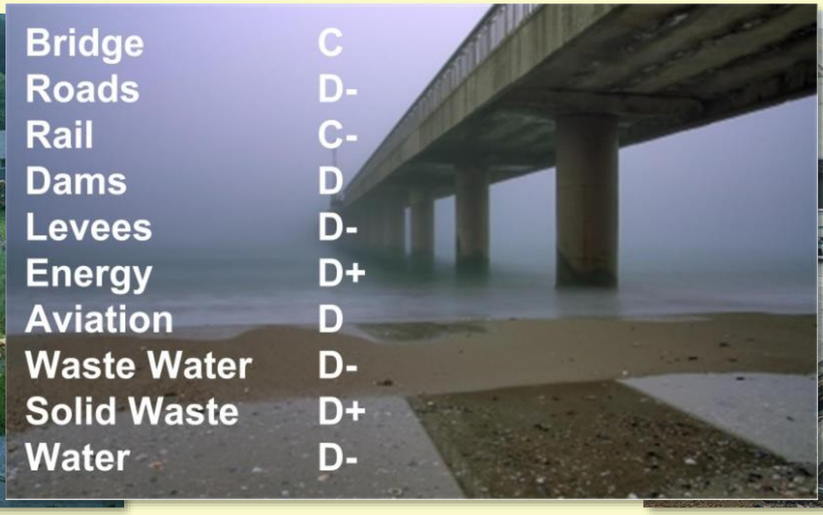
- Work reported here is joint with Carl Hewitt, Chris Houck, WooYoung Kim, Pohao Chang, Rajesh Karmani, Mark Astley, Dan Sturman, Stas Negara, Rajendra Panwar, Svend Frolund, Reza Shiftehfar, Koushik Sen among others.
- Supported in part by AFOSR/AFRL Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084 for the Assured Cloud Computing at the University of Illinois at Urbana-Champaign, and by the National Science Foundation under grant NSF CCF 14-38982 and NSF CCF 16-17401.

Motivation

INTERNET OF THINGS TO ENABLE SMART INFRASTRUCTURE

The Aging Civil Infrastructure

America's $20 trillion+ investment in civil infrastructure is in dire shape, and will continue to deteriorate if we fail to act.



Bridge	C
Roads	D-
Rail	C-
Dams	D
Levees	D-
Energy	D+
Aviation	D
Waste Water	D-
Solid Waste	D+
Water	D-

American Society of Civil Engineers report card grades

Continuous monitoring and precision targeting of maintenance can improve safety and save billions of dollars for infrastructure owners.

Sensor Clouds for Smart Infrastructure

- Smart infrastructure can improve safety, facilitate smart transportation systems.

Infrastructure is aging.
Estimated 70,000 structurally deficient bridges in US.

- Monitor infrastructure, pinpoint deficiencies.
- Control: dampen vibrations to limit damage
- *Scalable cyberphysical systems required*.

Keeping Tabs on the Infrastructure

"… one highly intelligent bridge knows what to do when trouble arises: send an e-mail."

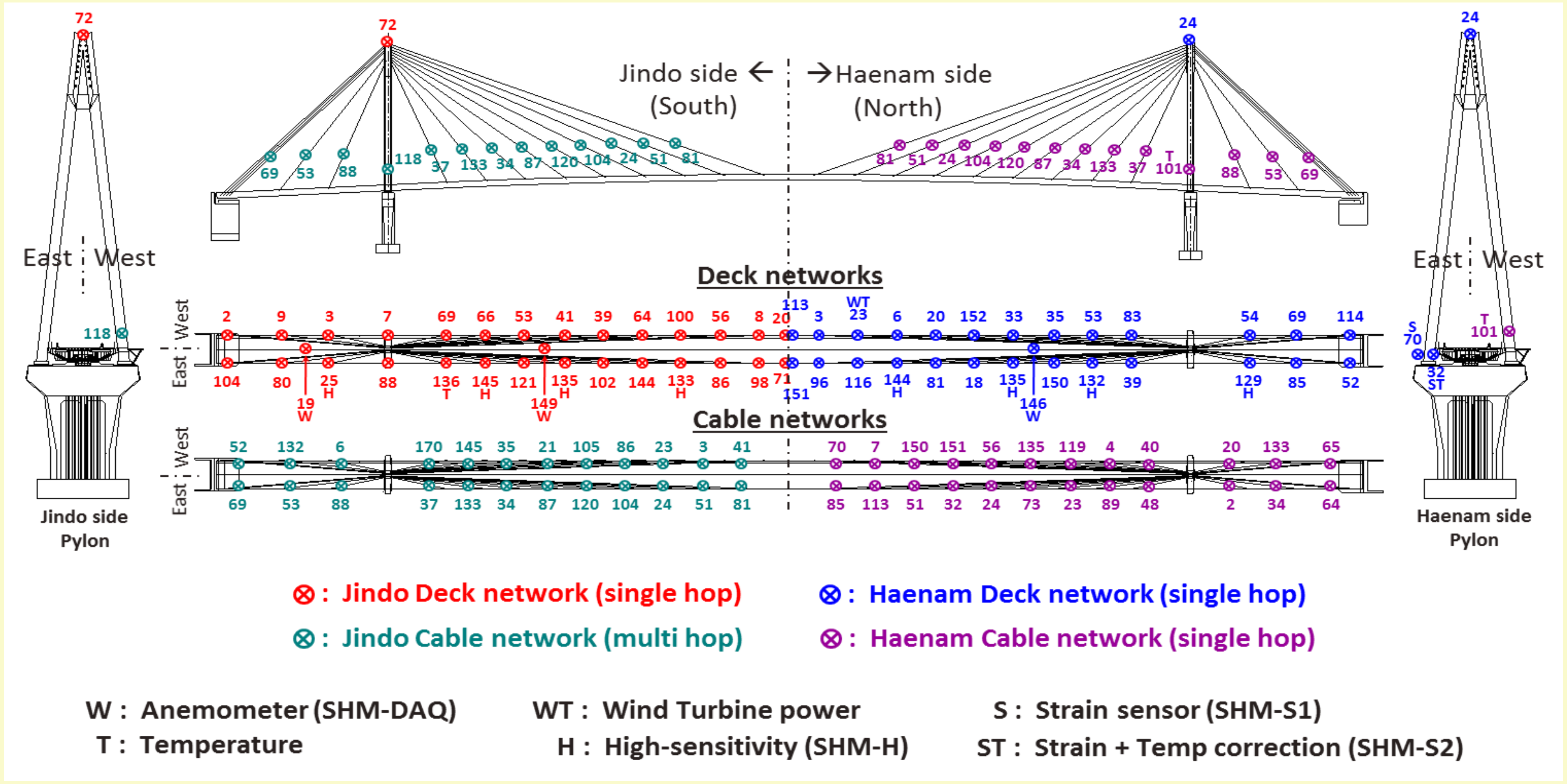
The New York Times

"A small army of electronic sentinels… monitor the bridge's structural health. (As of last week, the bridge said it was just fine.)"

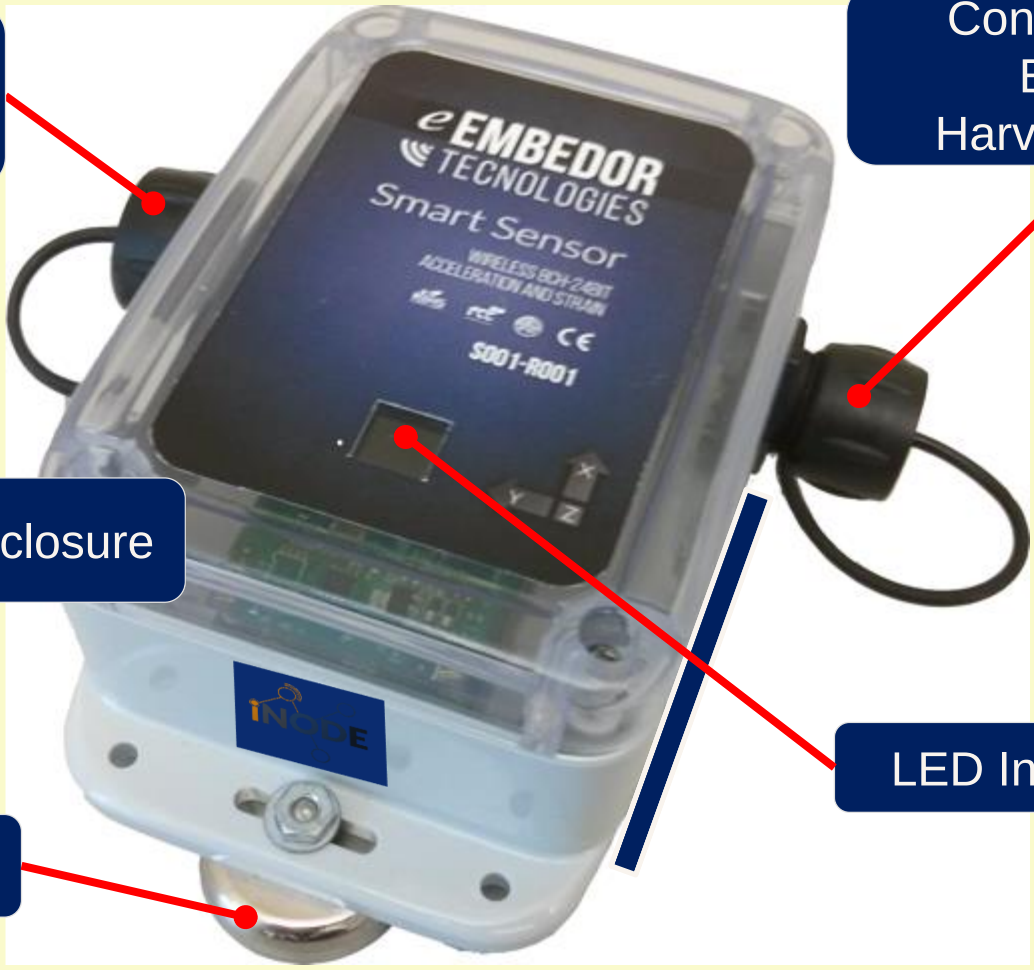
Illinois Structural Health Monitoring Project

See: http://shm.cs.illiniois.edu

Dense Sensor Clouds for Smart Structures


Jindo side ← (South)
→ Haenam side (North)
East West
Deck networks
Cable networks
Jindo side Pylon
Haenam side Pylon
⊗ : Jindo Deck network (single hop)
⊗ : Haenam Deck network (single hop)
⊗ : Jindo Cable network (multi hop)
⊗ : Haenam Cable network (single hop)
W : Anemometer (SHM-DAQ)
T : Temperature
WT : Wind Turbine power
H : High-sensitivity (SHM-H)
S : Strain sensor (SHM-S1)
ST : Strain + Temp correction (SHM-S2)

xnode: Environmentally Hardened Enclosure



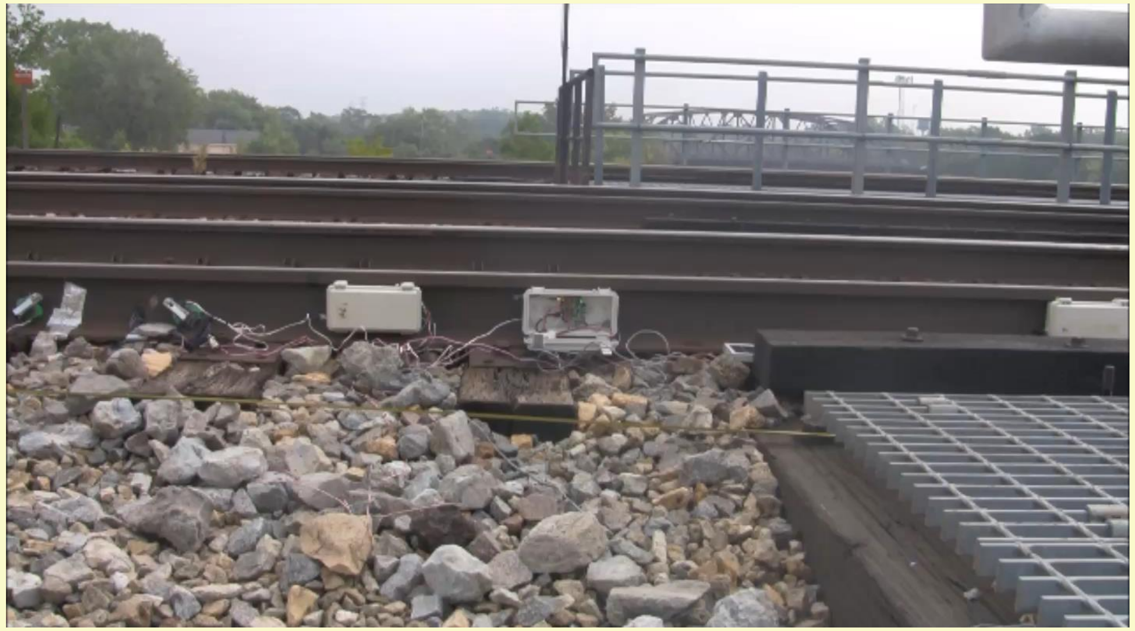
Connection to External Analog Sensors

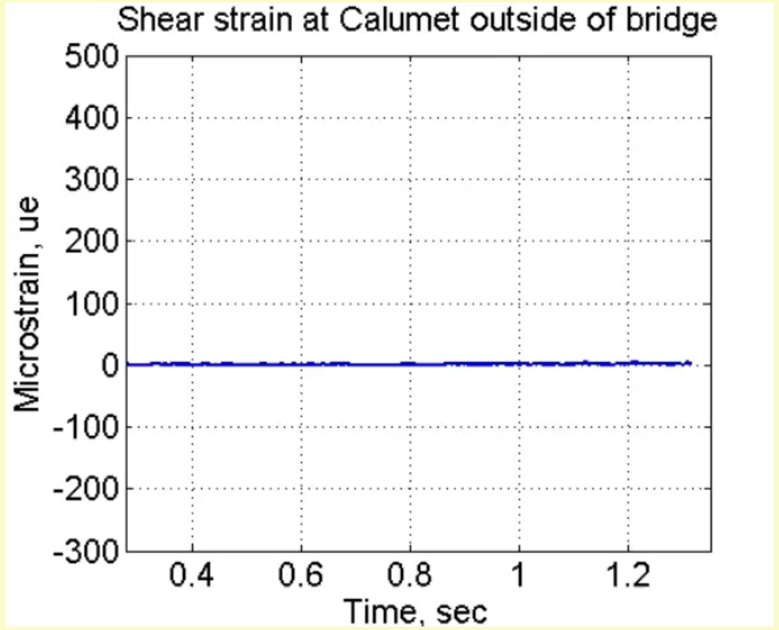
Connection to Energy Harvester/USB

IP66 Waterproof Enclosure

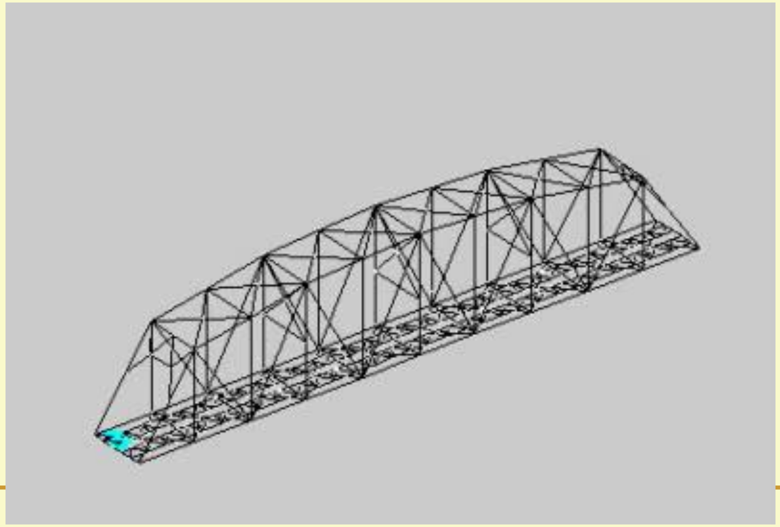
LED Indicator

Magnet

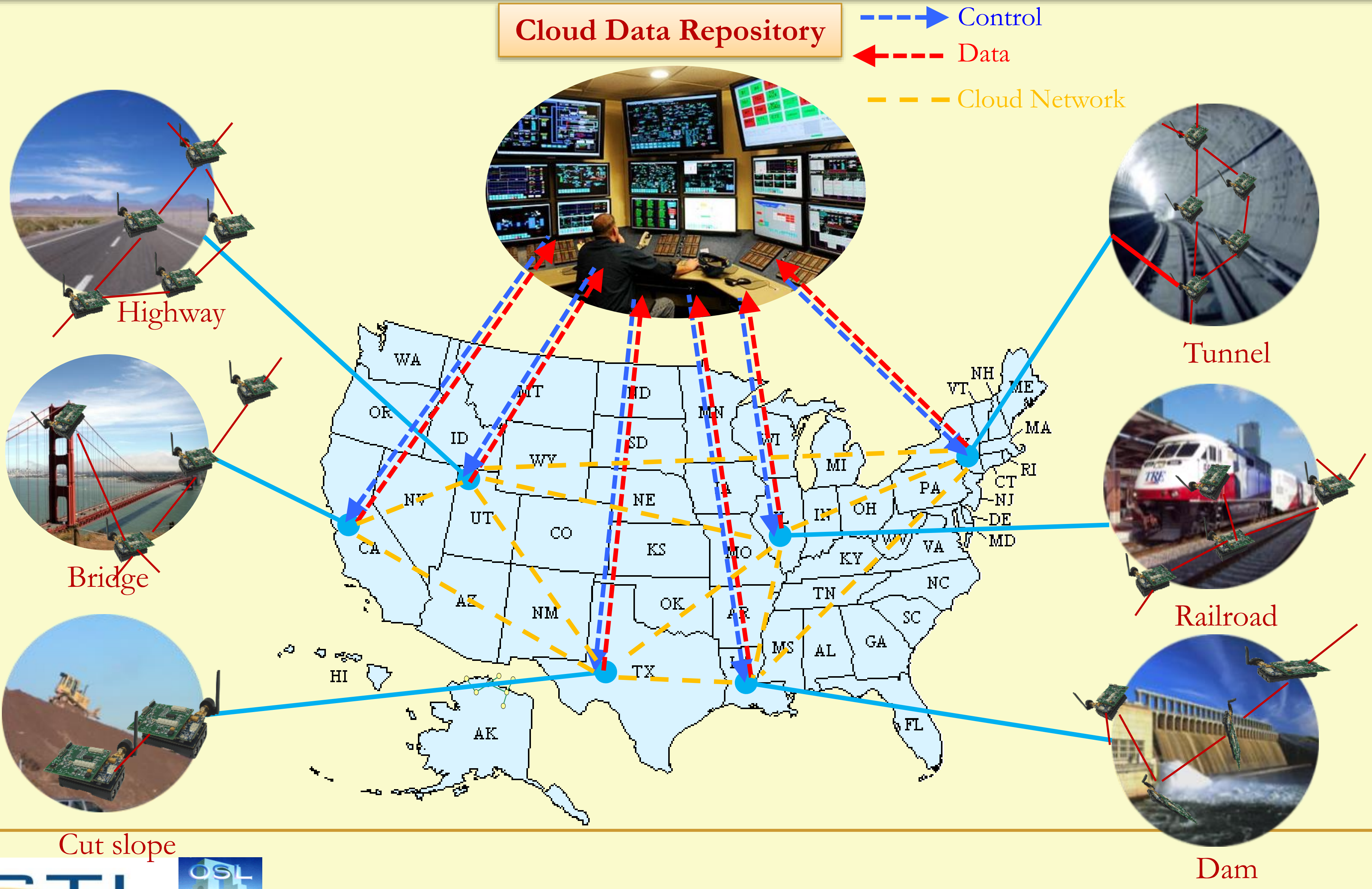




A calibrated model and strategically placed smart sensors allow for the force in all structural members to be determined with high accuracy and *remaining service life* to be estimated. Checkout the video at: http://embedortech.com



Example: National Smart Structures Grid



MODELS OF CONCURRENCY

Models of Concurrency

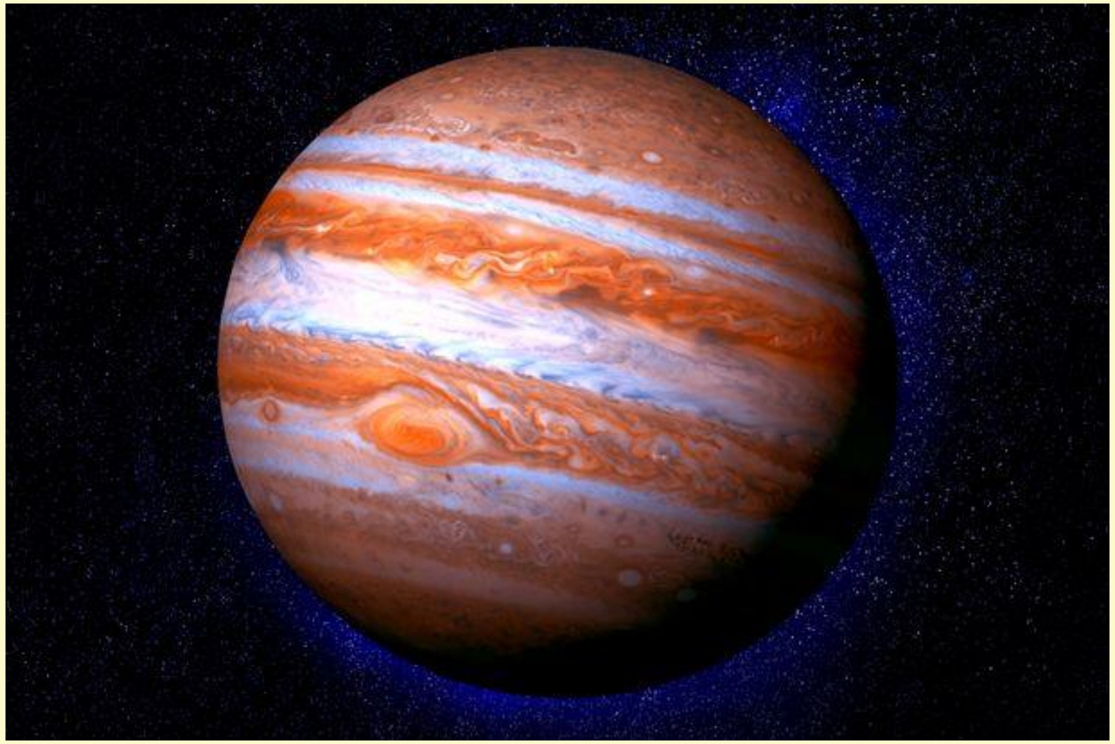
- Petri Nets
- Process Algebras
- Actors

Programming *Scalable* Applications

- Efficiency
- Distribution
- Concurrency
- Scalability
- Stability and Robustness

Scalable Applications

- Martian Rover (1990s)
- Twitter's message queuing
- LiftWeb Framework (Scala for web applications)
- Image processing in MS Visual Studio 2010
- Vendatta game engine (Erlang)
- Facebook Chat System (Erlang)
- LinkedIn
- Microsoft Orleans: used by >343 industries, platform for all of Halo 4 cloud services



***33** to 49 minutes for radio waves to travel from Jupiter to Earth*

Concurrency

``When people read about Scala, it's almost always in the context of concurrency. Concurrency can be solved by a good programmer in many languages, but it's a tough problem to solve. Scala has an *Actor library* that is commonly used *to solve concurrency problems*, *and it makes that problem a lot easier to solve*."

--Alex Payne, ``*How and Why Twitter Uses Scala*"

http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html

(emphasis added)

Stability and Scalability

“..the *actor model has worked really well* for us, and we wouldn't have been able to pull that off in C++ or Java. Several of us are big fans of Python and I personally like Haskell for a lot of tasks, but the bottom line is that, while those languages are great general purpose languages, none of them were designed with the actor model at heart.”

--Facebook Engineering

https://www.facebook.com/notes/facebook-engineering/chat-stability-and-scalability/51412338919

Actor Languages and Frameworks

- Erlang
- E
- Axum
- Stackless Python
- Theron (C++)
- RevActor (Ruby)
- Dart
- Asynchronous Agents Library
- Scala Actors/Akka
- ActorFoundry
- SALSA
- Kilim
- Jetlang
- Actor’s Guild
- Clojure
- … a growing list

Characteristics of the Actor Model

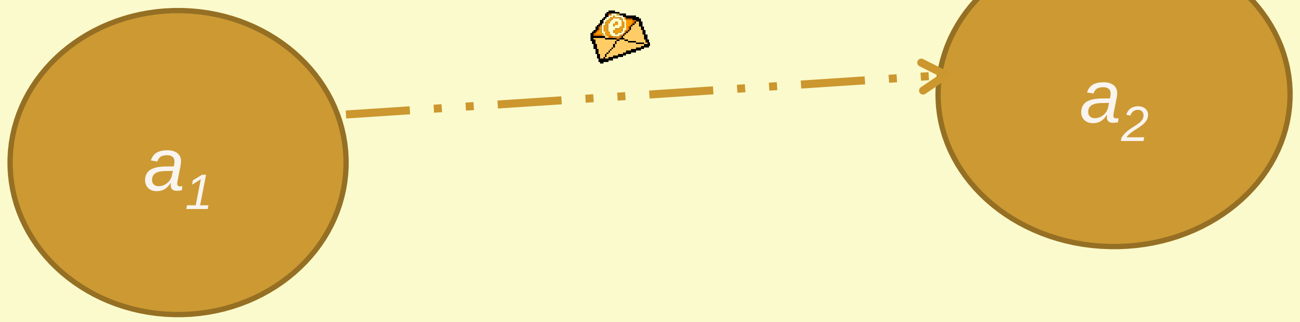
Computation broken into autonomous, concurrent agents called *actors*:

- Actors do not share state
  - Analogous to animals in natural systems.
- Each actor operates asynchronously
  - The rate at which an actor operates may vary.
  - An actor is like a *virtual processor.*
- An actor may interact with other actors.

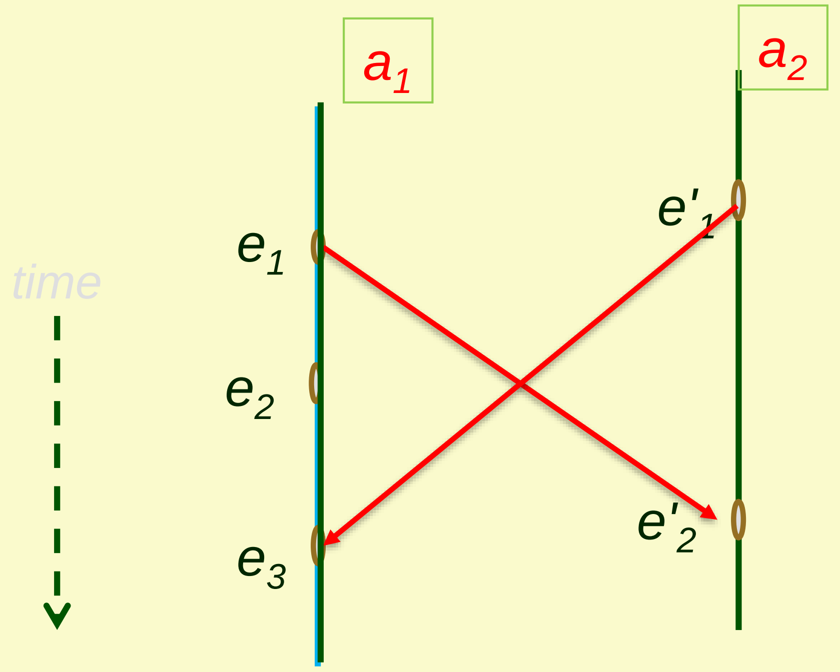
[Actors: A Model of Concurrent Computation in Distributed Systems," Gul Agha. MIT Press, 1986.]

Message-Passing

- There is no action at a distance
- An actor a_1 an only affect a_2 by sending it a message.
- Messages are asynchronous



Distribution and Parallelism



- Each actors represents a point in a virtual space.
- Events at an actor are ordered linearly.
- Events may change the state of an actor
- An event on one actor may activate an event on another by sending a message (causal order).
- Transitive closure results in a partial order

Fairness

- Each actor makes progress if it can:
  - *If multiple actors execute on a single processor, each actor is scheduled.*
- Every messages is eventually delivered if it can be:
  - *When an actor is idle and has a pending message, it processes that message.*
  - *Multiple pending messages are processed in an order so none is permanently ignored by the target actor.*

Actor Names

- The *name (mail address)* of each actor is unique and cannot be guessed.
- An actor must know the *name (mail address)* of the target actor to send it a message
  - Called the *locality property* of actors.
- Locality property provides a built in *capability architecture* for security.

Actor Topology

- If an actor a_1 knows the address of another actor a_2 , a_1 may communicate the name of an a_2 in a message.
  - The interconnection topology of actors is dynamic.
- Supports mobility and reconfiguration of actors.

Actor Creation

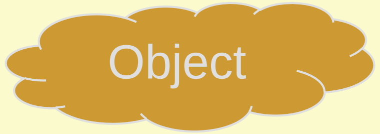
New actors may be created:

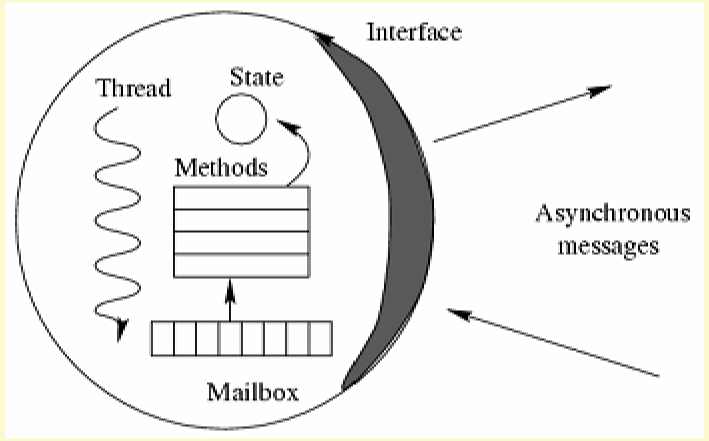
- Increases the available concurrency in a computation.
- Facilitates dynamic parallelism for load balancing.
- Enables mechanisms for fault-tolerance.

Actor anatomy: Actors and Threads

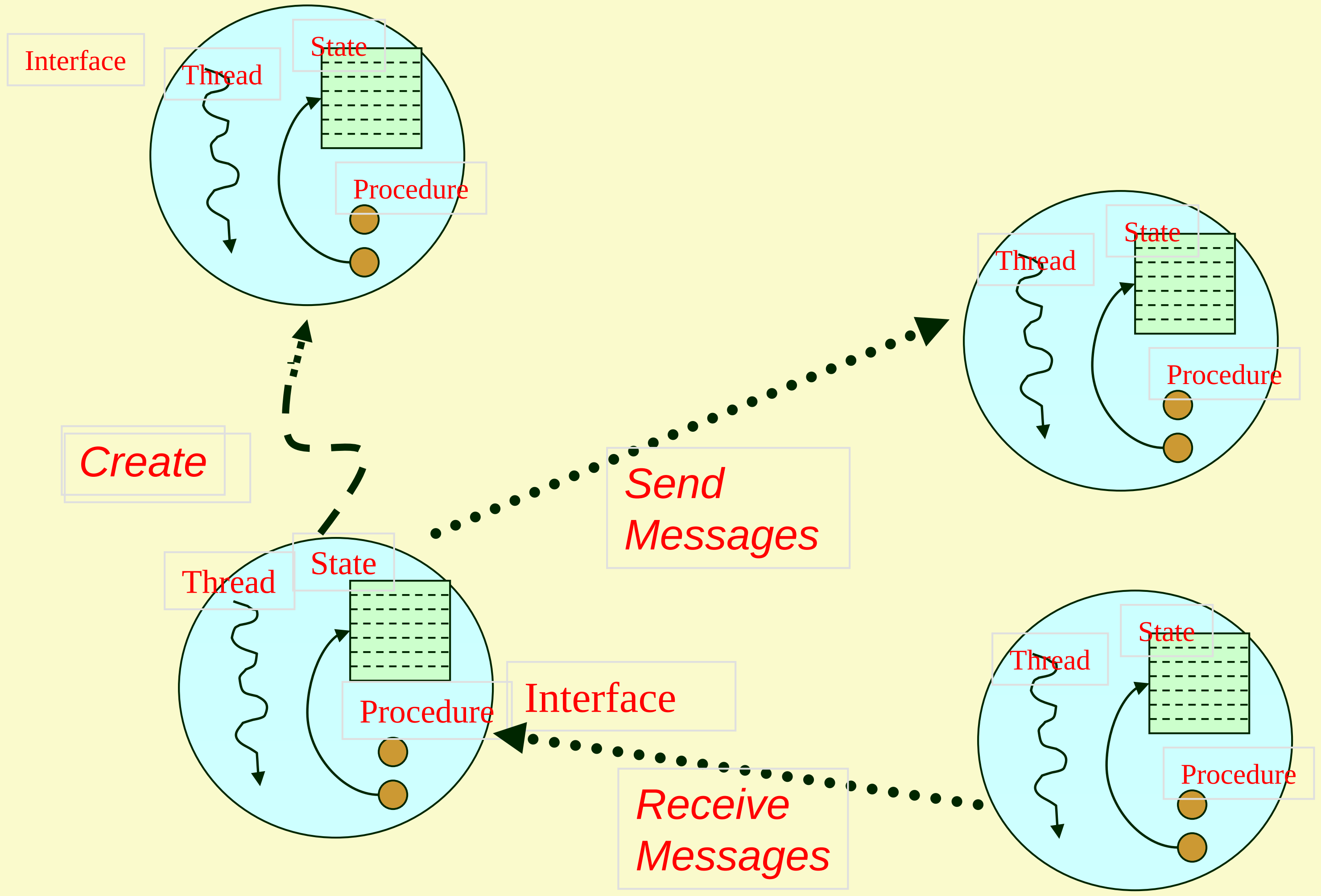
Actors = encapsulated state + behavior +

independent control + mailbox





The Actor Model: Runtime Support



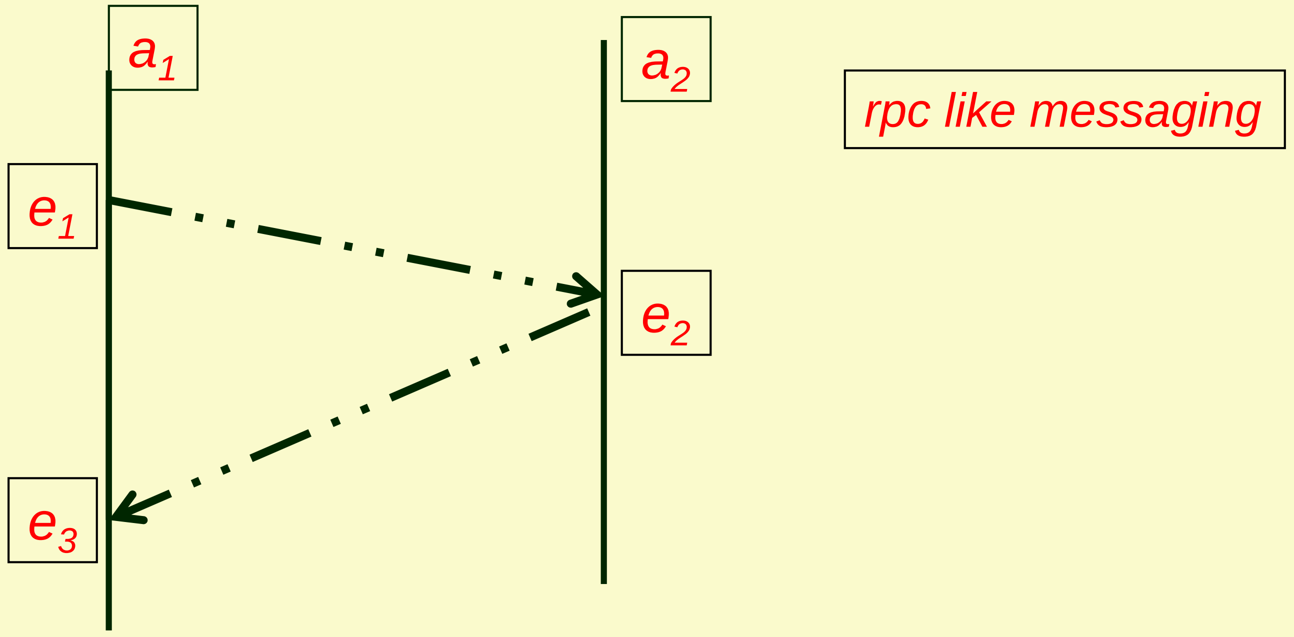
Defining an actor language

Start with a sequential object-based language or framework, add concurrency to objects, operators for:

- actor creation
  - create(class, params)
  - *Locally or at remote nodes*
- message sending
  - send(actor, method, params)
- state change
  - ready to process next message

Message Patterns

- More complex message patterns may be defined in terms of asynchronous messages:



Actor Event Diagrams

Actor Encapsulation: State Isolation

- Recall: *no shared state* between actors
- ‘Access’ another actor’s state *only* by sending it a message and requesting it:
  - Messages have send-by-value semantics
  - Implementation may be relaxed on shared memory platforms, if “safe”

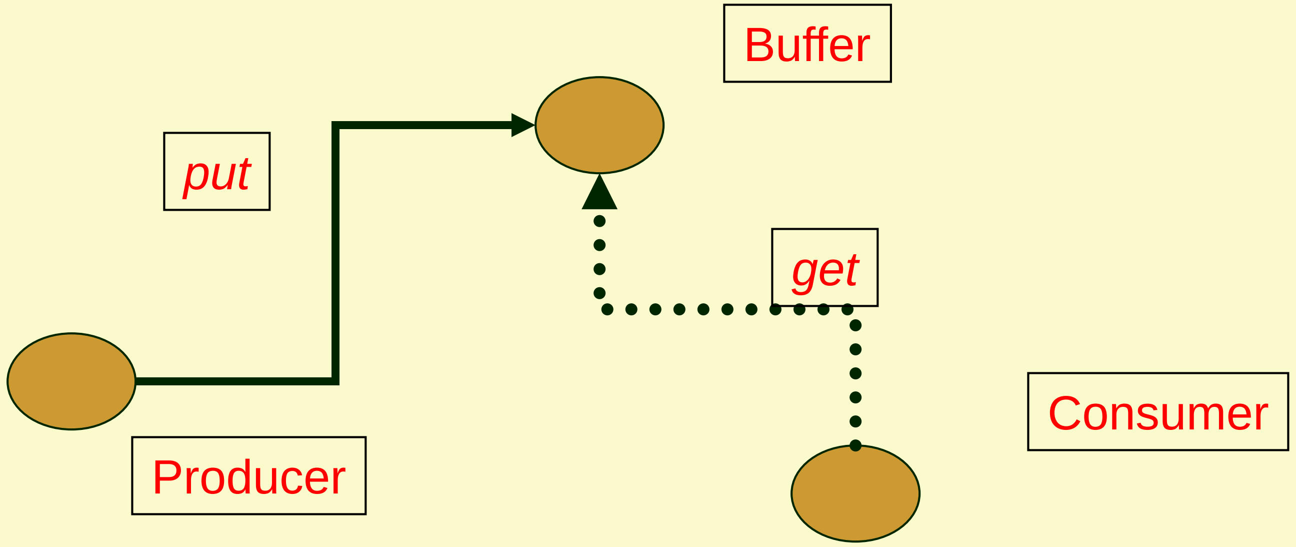
Location Transparent Naming

- Enables *automatic* load-balancing and fault-tolerance mechanisms
  - Run-time can exploit resources available on cluster, grid or scalable multicores (distributed memory)
- Uniform model for multicore and distributed programming

Synchronization and Coordination

- Essential for correct functioning of actor systems
- A source of complexity in concurrent programs

Synchronizing in a Concurrent World



- The interface of an actor may be dynamic:
  - Cannot get from an empty buffer
  - Cannot put into a full buffer

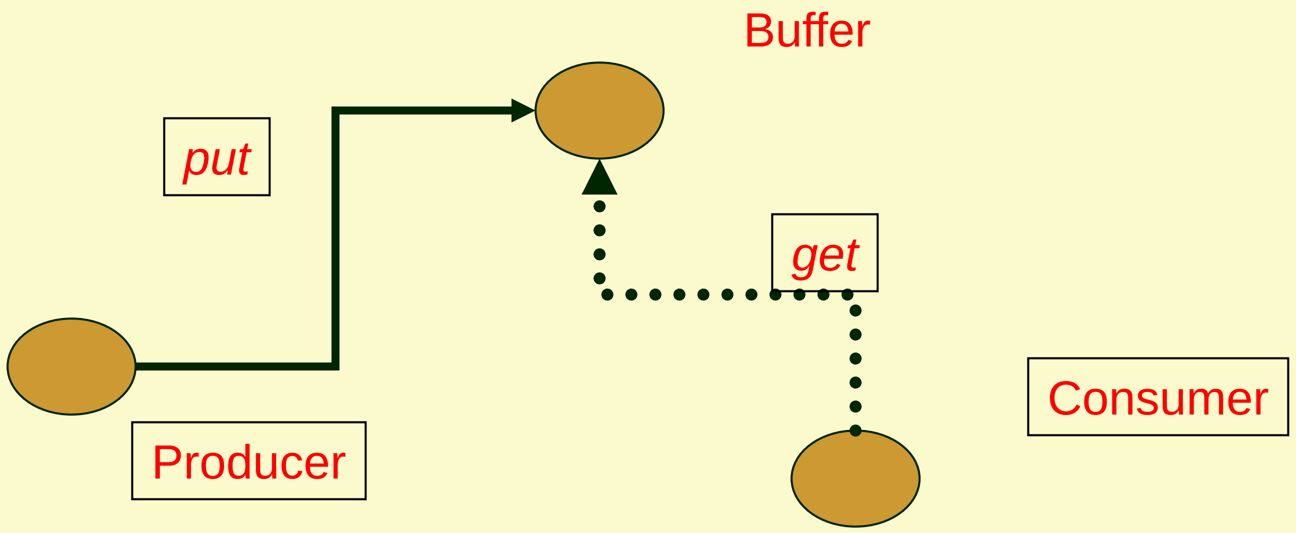
Separation of Concerns

- Abstract Data Types:
  - Enable separation of *interface (what)* from the *representation (how).*
- Actors:
  - *When* actions happen is underspecified (*asynchrony).*
  - Recipient may not be ready to process a message when it arrives – *synchronization constraints (when).*
  - Separate specification of *when* from *how* to facilitate modularity in code.

Local Synchronization Constraints

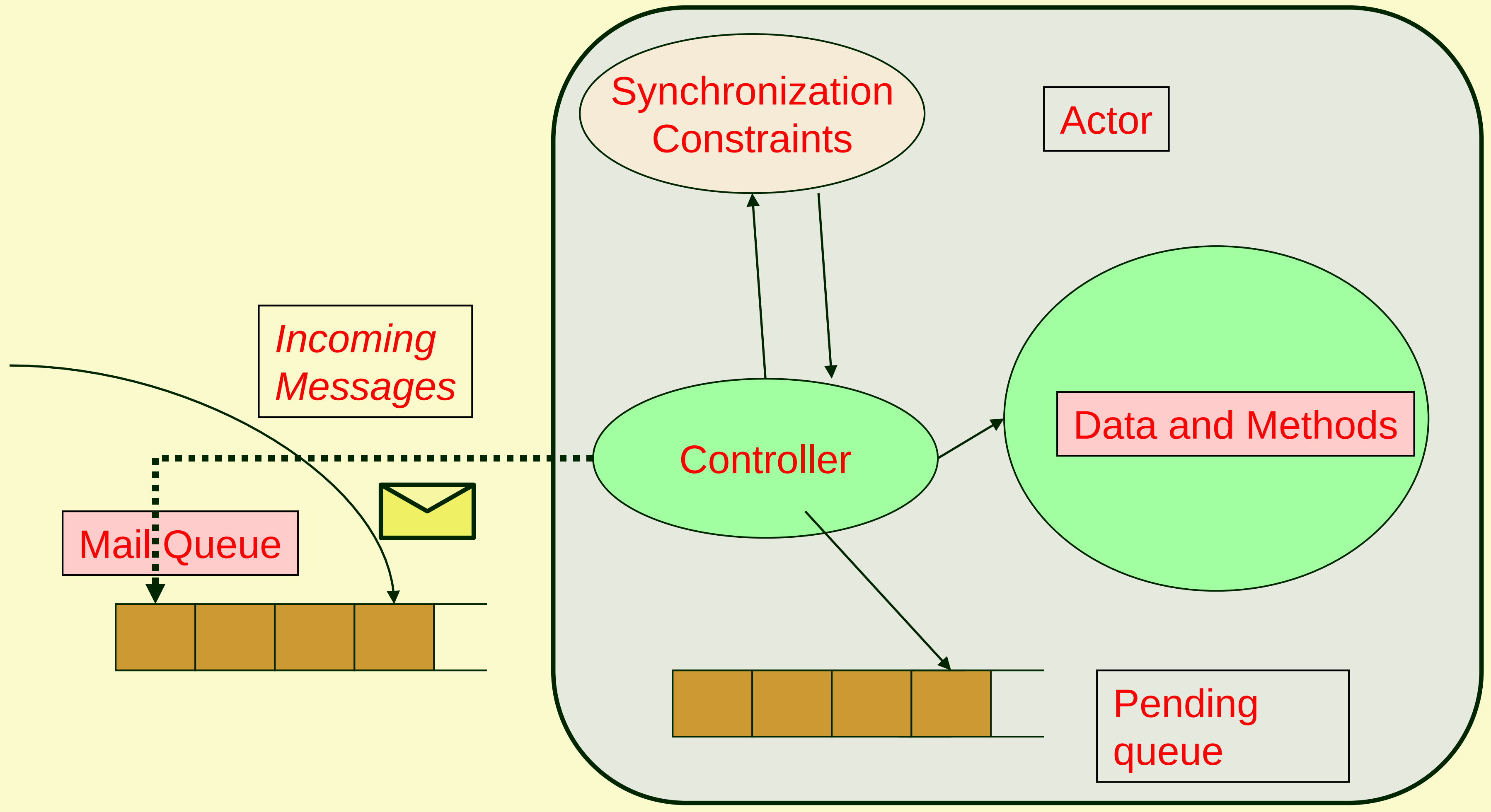
- Constrain the “local” order of processing messages
  - Delay or reject out of order messages
  - Function of local state and message contents
- These have *delay* semantics i.e. disabled messages are buffered
- Implementations: Disabling constraints in AF, Pattern matching in Erlang, Scala

Expressing Local Synchronization Constraints (Abstractly)



- Per actor logical rules which determine the legality of invocations:
  - disable get when empty? (buffer)

Implementation of Local Synchronization Constraints



Scalable Reasoning Tools

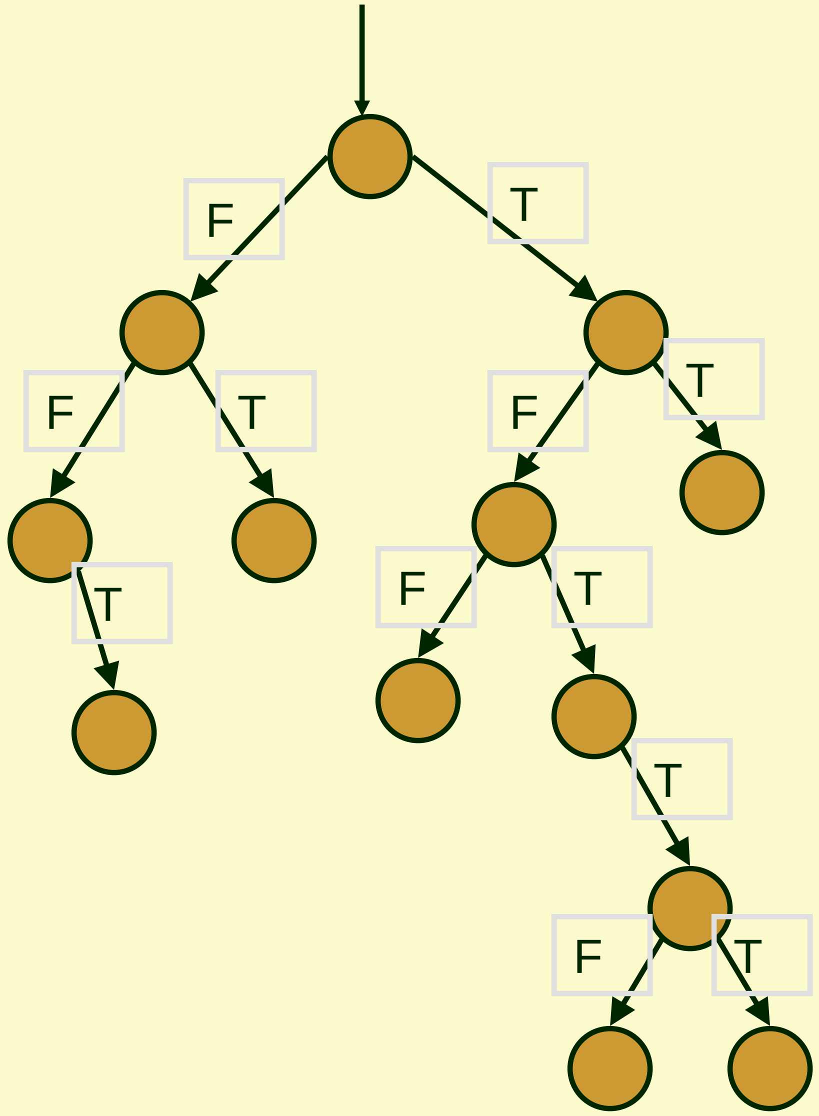
- Computational Learning for Verification
- Concolic Testing and its variants
- Runtime verification (Monitoring)
- Inferring interfaces: session types, concurrency structure
- Computational learning for verification (won’t discuss today)

Quantitative Tools:

- Statistical Model Checking
- Euclidean Model Checking

Execution Paths of a Program

- Can be seen as a binary tree with possibly infinite depth
  - Computation tree
- Each node represents the execution of a "if then else" statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs.
- *What about loops?*
  - *Unroll to finite depth.*



What is Testing?

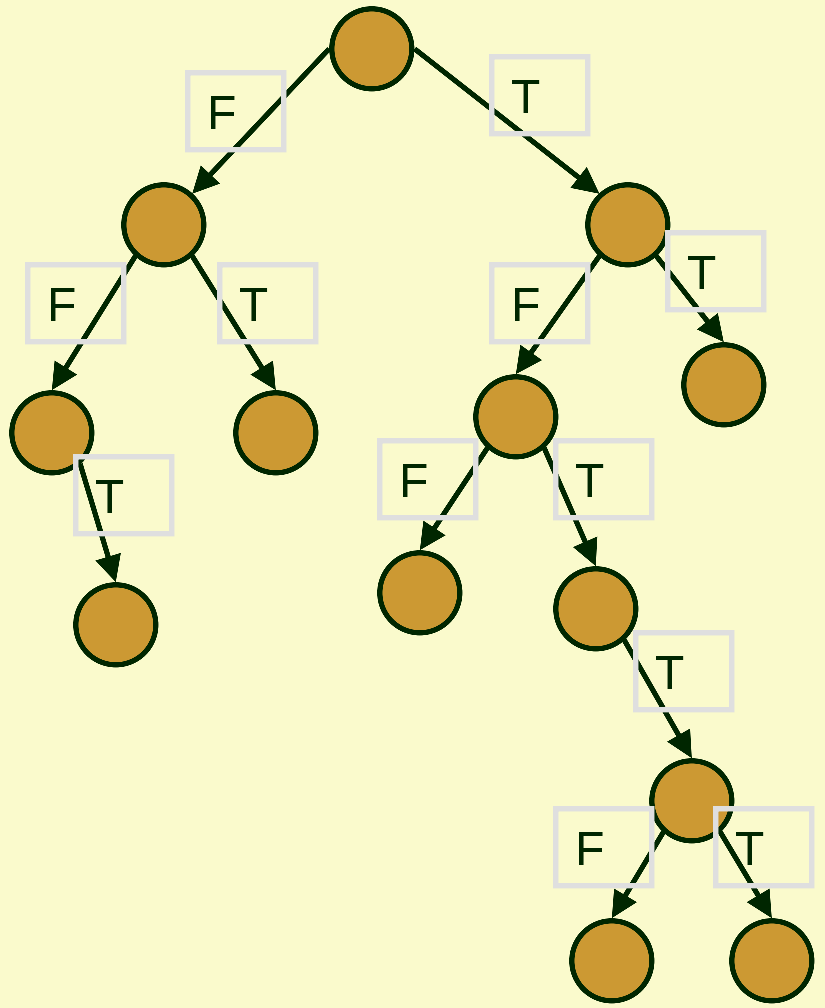
- Execute the program and observe how it behaves under different scenarios:
  - Vary the inputs.
  - In concurrent programs: vary the schedules.

Goal

- Automated Scalable Unit Testing of real-world sequential programs
  - Generate test inputs
  - Execute unit under test on generated test inputs
    - so that all reachable statements are executed
  - Any assertion violation gets caught

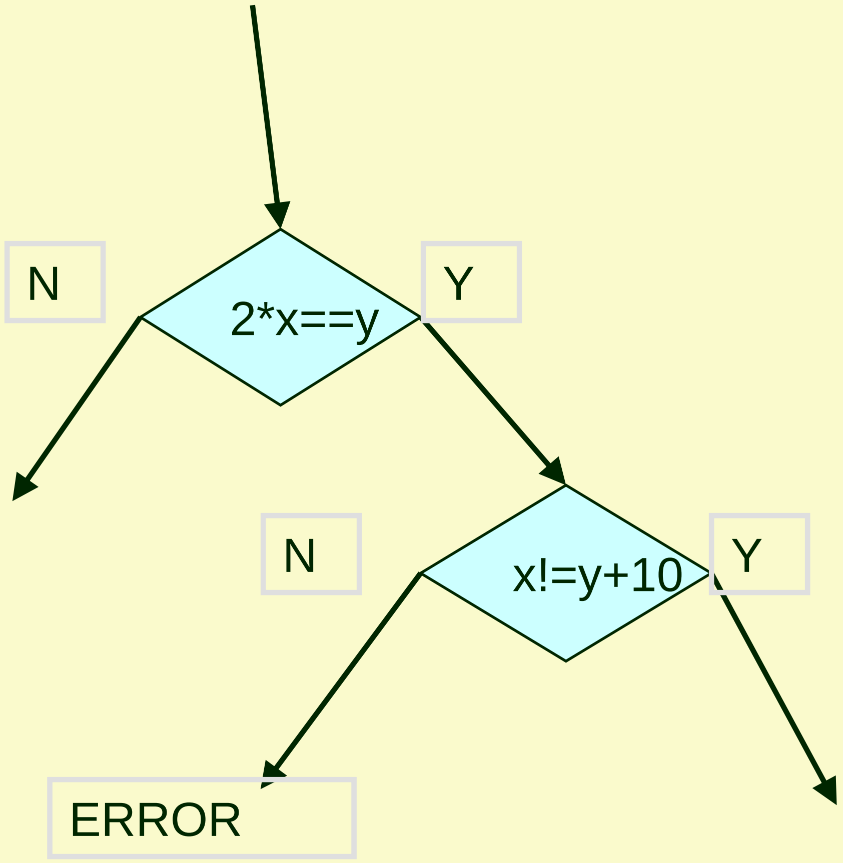
What is a bug?

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors



Example of Computation Tree

```
void test_me(int x, int y) {
  if(2*x==y){
    if(x != y+10){
         printf("I am fine here");
    } else {
         printf("I should not reach here");
         ERROR;
    }
  }
}
```



Random Testing

- generate random inputs
- execute the program on generated inputs
- Probability of reaching an error can be astronomically less

```
test_me(int x){
   if(x==94389){
        ERROR;
   }
}
```

Probability of hitting ERROR = $1/2^{32}$

Symbolic Execution

- use symbolic values for input variables
- execute the program symbolically on symbolic input values
- collect symbolic path constraints
- use theorem prover to check if a branch can be taken

```
test_me(int x){
   if (x==94389){
        ERROR;
   }
}
```

Symbolic Execution

- What if we can solve the constraint?
- Symbolic execution will say both branches are reachable:

  False positive
- Does not scale for large programs

```
test_me(int x){
    if((x%10)*4!=17){
        ERROR;
    } else {
        ERROR;
    }
}
```

Approach

- Combine concrete and symbolic execution for unit testing
  - **Conc**rete + Symb**olic** = Concolic
- In a nutshell
  - Use concrete execution over a concrete input to guide symbolic execution
  - Concrete execution helps Symbolic execution to simplify complex and unmanageable symbolic expressions
    - by replacing symbolic values by concrete values
- Achieves Scalability
  - Higher branch coverage than random testing
  - No false positives or scalability issue like in symbolic execution based testing

Example: Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7	$x = x_0, y = y_0$	

Example : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$(y_0 * y_0)\%50 \; != x_0$
x = 22, y = 7, z = 49	$x = x_0$, $y = y_0$, $z = (y_0 * y_0)\%50$	

Solve: $(y_0 * y_0)\%50 == x_0$
Don't know how to solve!
Stuck?

Example : Simultaneous Concrete and Symbolic Execution

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

Solve: foo $(y_0) == x_0$
Don't know how to solve!
Stuck?

```
void testme (int x, int y) {

    z = foo (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

$\text{foo}(y_0) \; != x_0$

x = 22, y = 7,
z = 49

$x = x_0$, $y = y_0$,
$z = \text{foo}(y_0)$

Example : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
     return (v*v) % 50;
}

void testme (int x, int y) {
     z = foo (y);
     if (z == x) {
          if (x > y+10) {
               ERROR;
          }
     }
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
$x = 22, y = 7, z = 49$	$x = x_0, y = y_0, z = (y_0 * y_0)\%50$	$(y_0 * y_0)\%50 \,!= x_0$

Solve: $(y_0 * y_0)\%50 == x_0$
Don't know how to solve!
Not Stuck!
Use concrete state
Replace y_0 by 7 (sound)

Example : Simultaneous Concrete and Symbolic Execution

Concrete Execution

Symbolic Execution

```
int foo (int v) {
      return (v*v) % 50;
}

void testme (int x, int y) {
      z = foo (y);
      if (z == x) {
            if (x > y+10) {
                  ERROR;
            }
      }
}
```

concrete state | symbolic state | path condition

Solve: $49 == x_0$
Solution : $x_0 = 49$, $y_0 = 7$

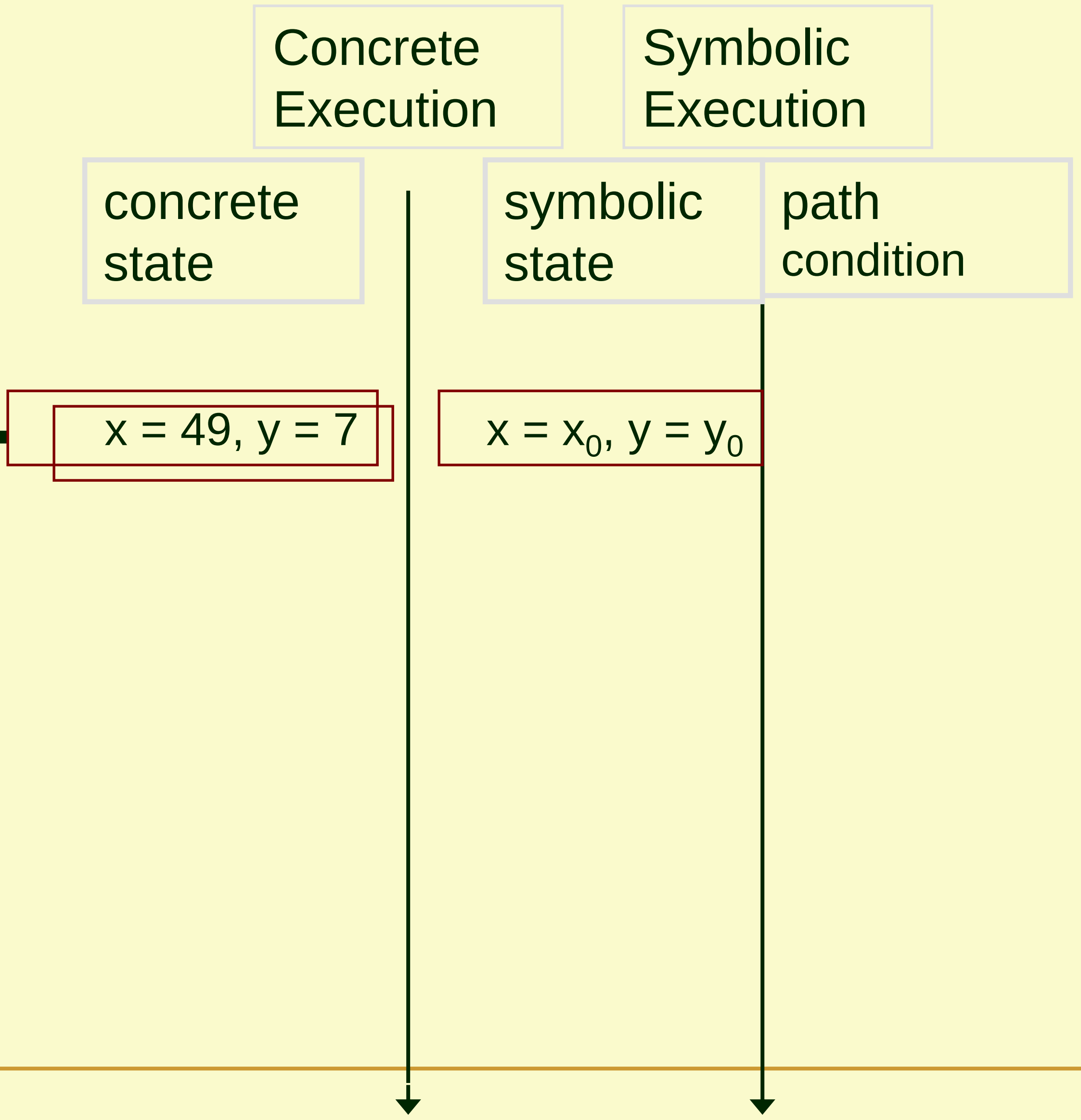
$49 \; != x_0$

$x = 22$, $y = 7$, $z = 48$

$x = x_0$, $y = y_0$, $z = 49$

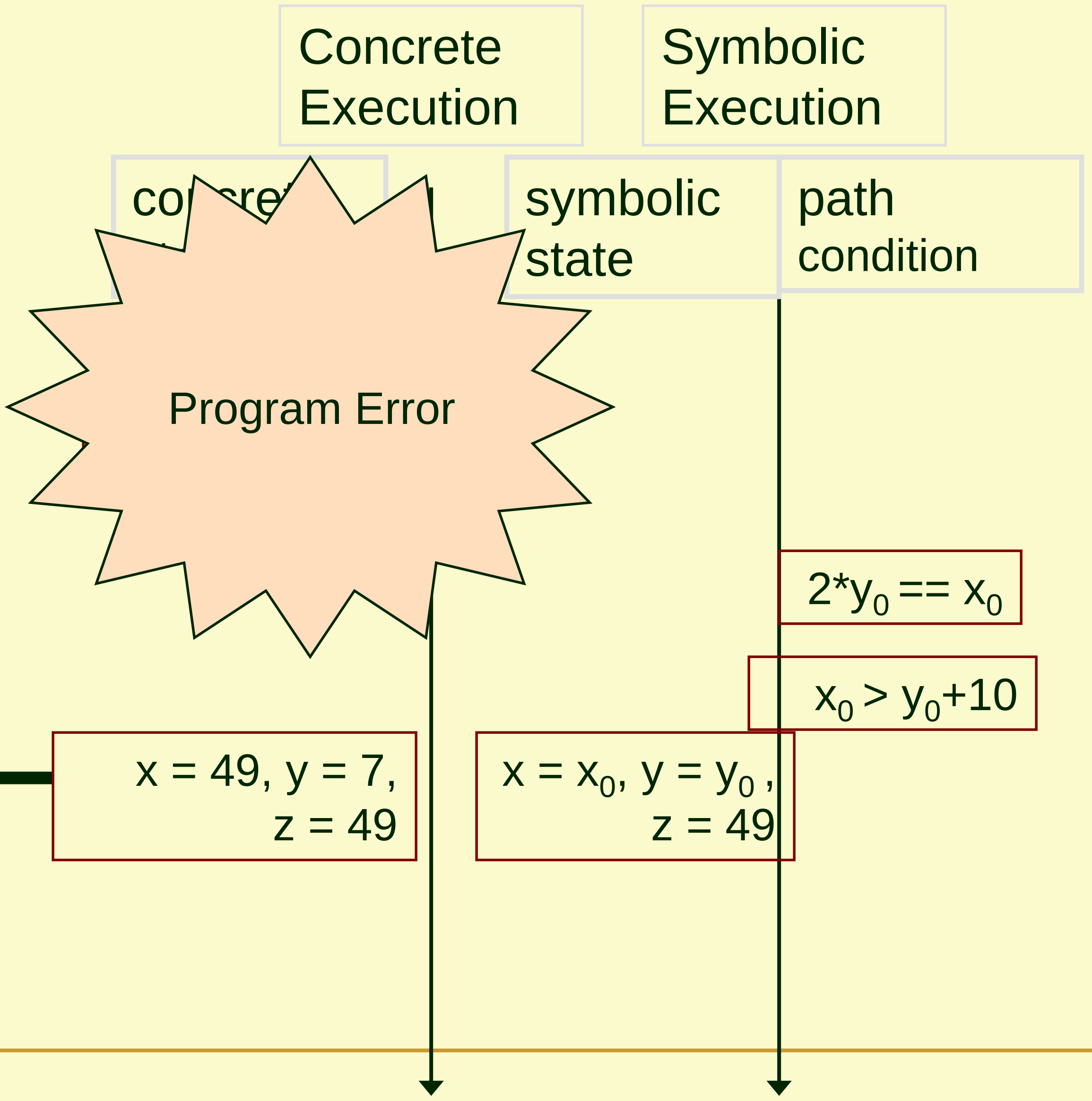
Example : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
      return (v*v) % 50;
}

void testme (int x, int y) {
      z = foo (y);
      if (z == x) {
            if (x > y+10) {
                  ERROR;
            }
      }
}
```

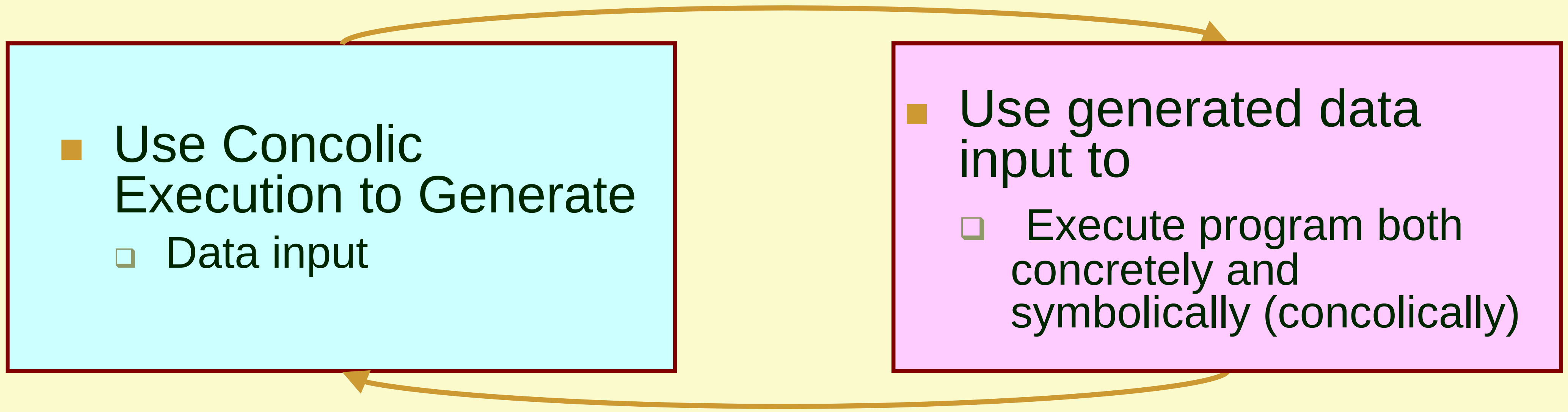


Example : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```



Concolic Testing in a Nutshell



- Use Concolic Execution to Generate
  - Data input

- Use generated data input to
  - Execute program both concretely and symbolically (concolically)

- Use concrete execution to Guide symbolic execution
- Use smart search strategies: e.g. *Concolic Walk* in space defined by constraints

- Use symbolic execution to
  - To generate data input

Concolic Execution for Concurrent Programs

- Schedules are another branching condition
- Partial order reduction helps
  - Multistep Semantics for Actors
- Still too many interleavings..
  - Use backward symbolic execution
  - Branch coverage is an uninteresting metric
- Unchecked conditions
  - Runtime Verification

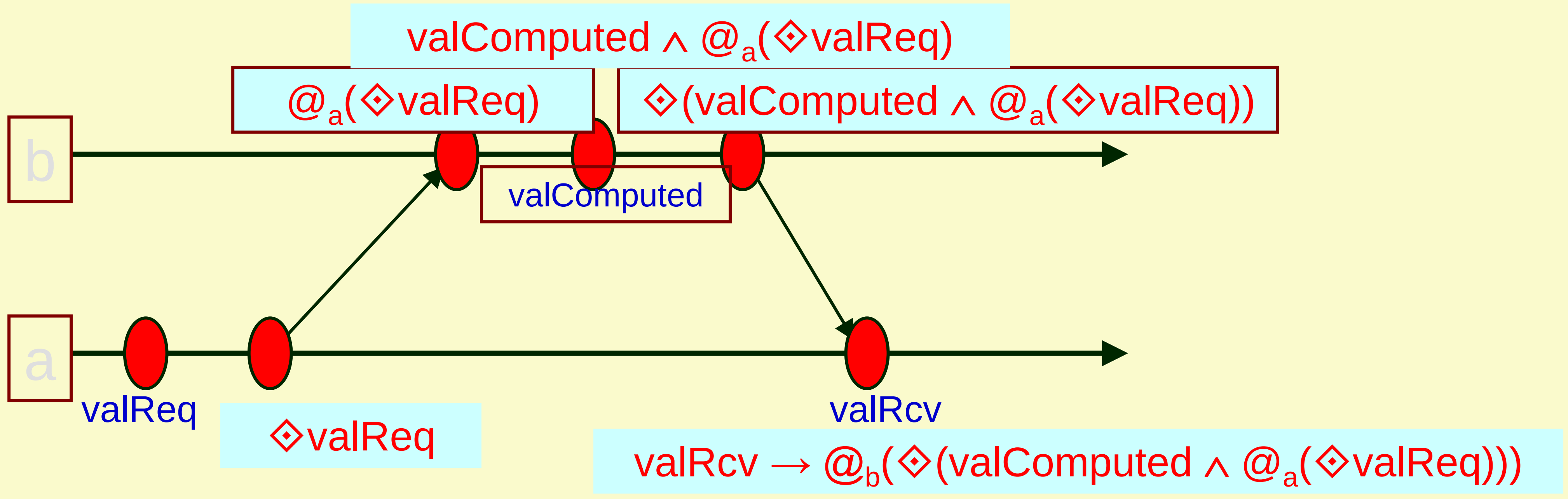
Decentralized Runtime Verification

- Properties expressed with respect to an actor (Epistemic Logic)
- Properties are in Distributed Temporal Logic
- Decentralize Monitoring
  - Maintain knowledge of relevant state at each process
  - Update knowledge with incoming messages
  - Attach knowledge with outgoing messages
  - At each actor check safety property against local knowledge

Decentralized Monitoring Example

"If a receives a value from b then b calculated the value after receiving request from a"

$\text{valRcv} \rightarrow @_b(\diamond(\text{valComputed} \wedge @_a(\diamond\text{valReq})))$



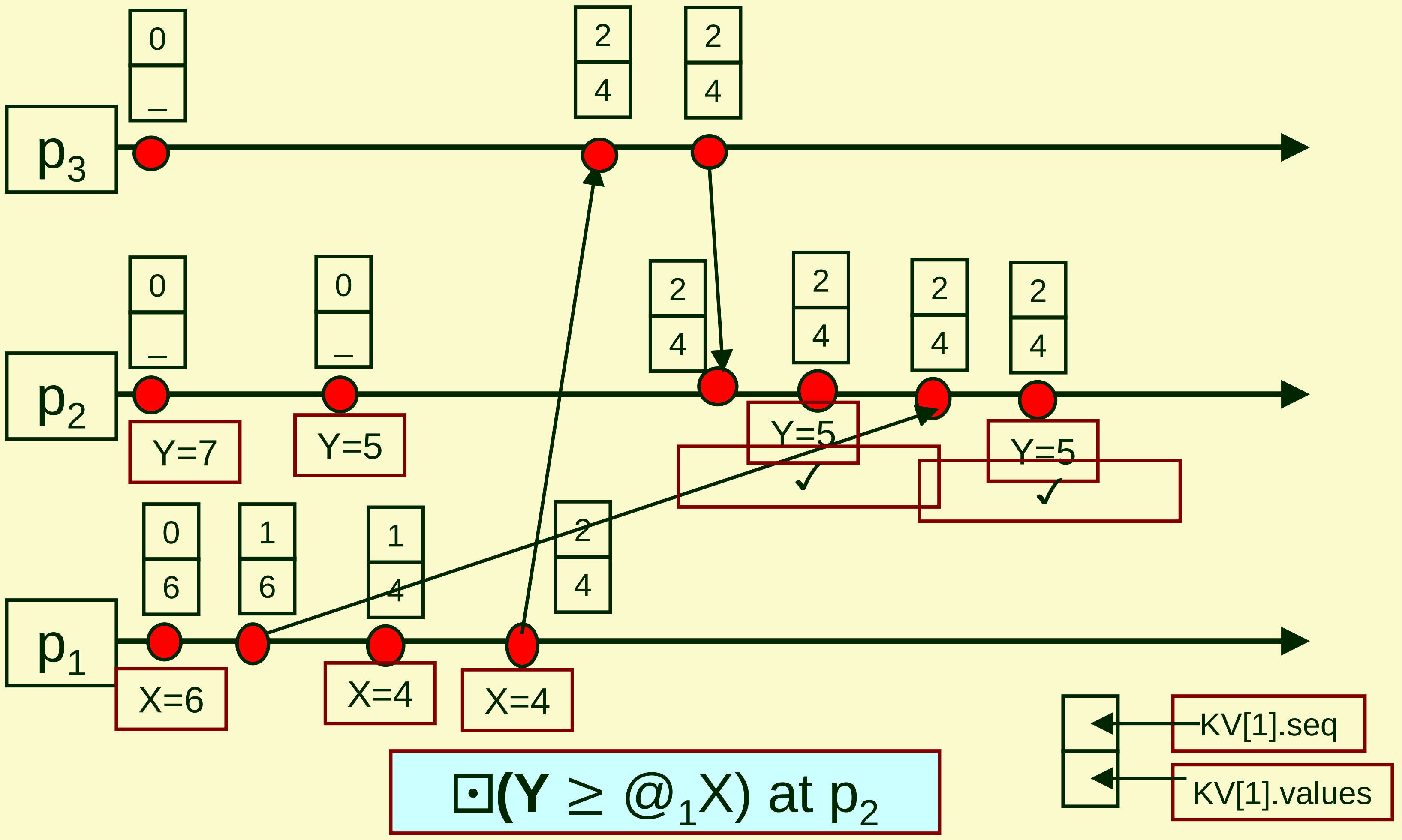
valComputed ∧ @a(◇valReq)
@a(◇valReq)
◇(valComputed ∧ @a(◇valReq))
b
valComputed
a
valReq
◇valReq
valRcv
valRcv → @b(◇(valComputed ∧ @a(◇valReq)))

KnowledgeVector

Let KV be a vector. KV is a knowledge vector if it has:

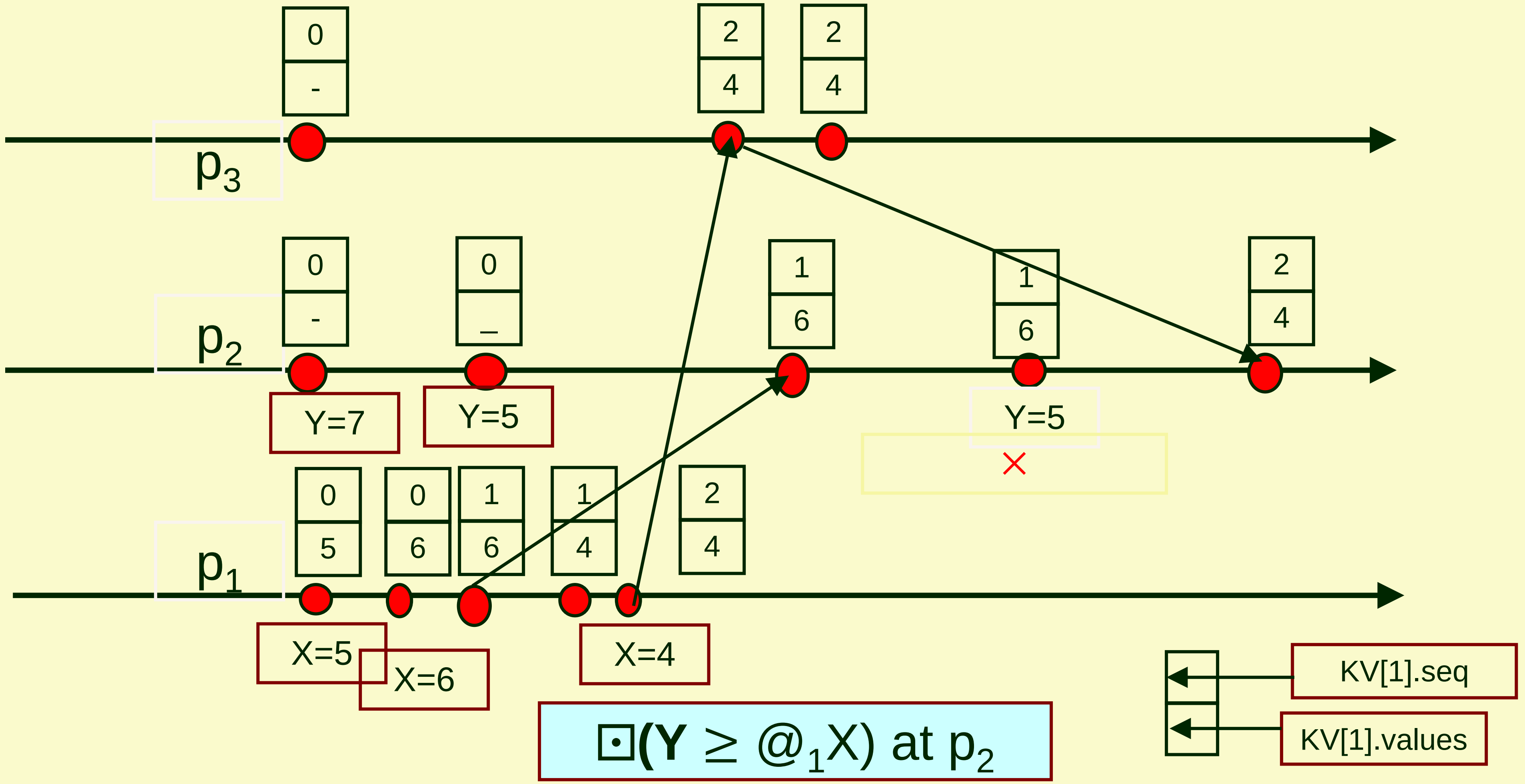
- one entry for each process appearing in formula
- KV[j] denotes entry for actor j
- KV[j].seq is the sequence number of last event seen at actor j
- KV[j].values stores values of j-expressions and j-formulae

Example



$\boxdot(\mathbf{Y} \geq @_1 X)$ at p_2

Example: Another Potential Execution

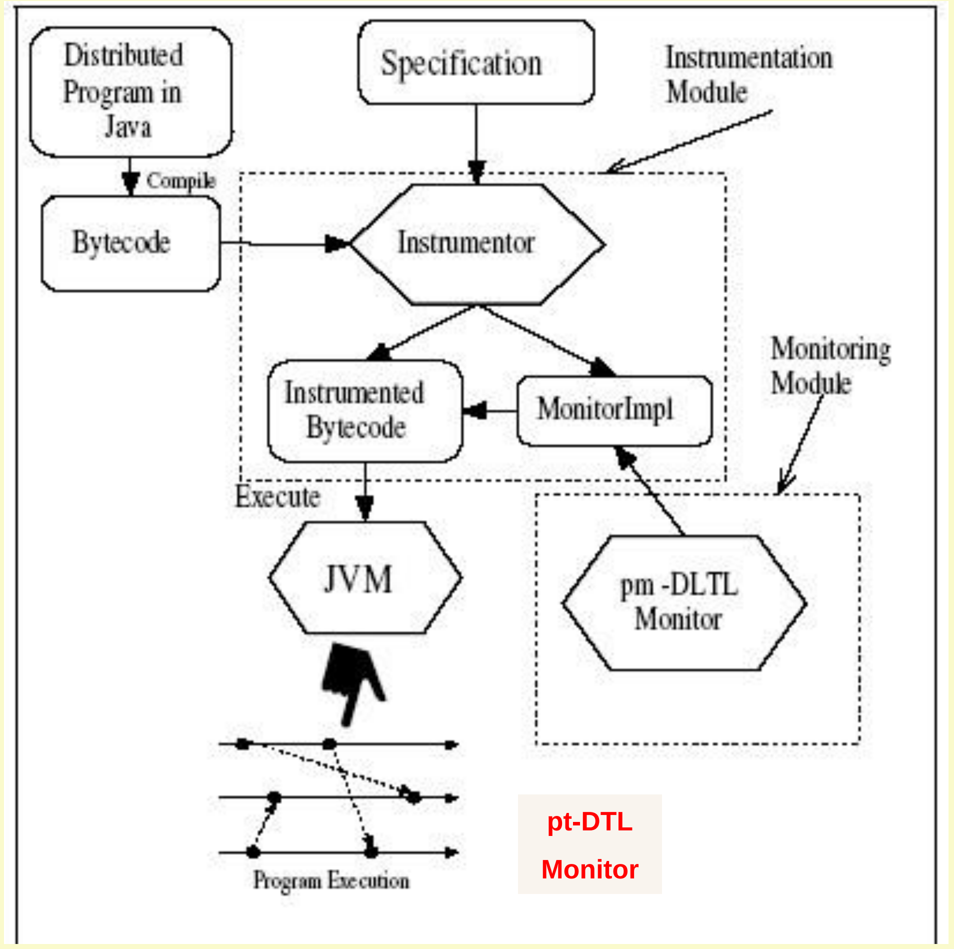


$\boxdot(\mathbf{Y} \geq @_1X)$ at p_2

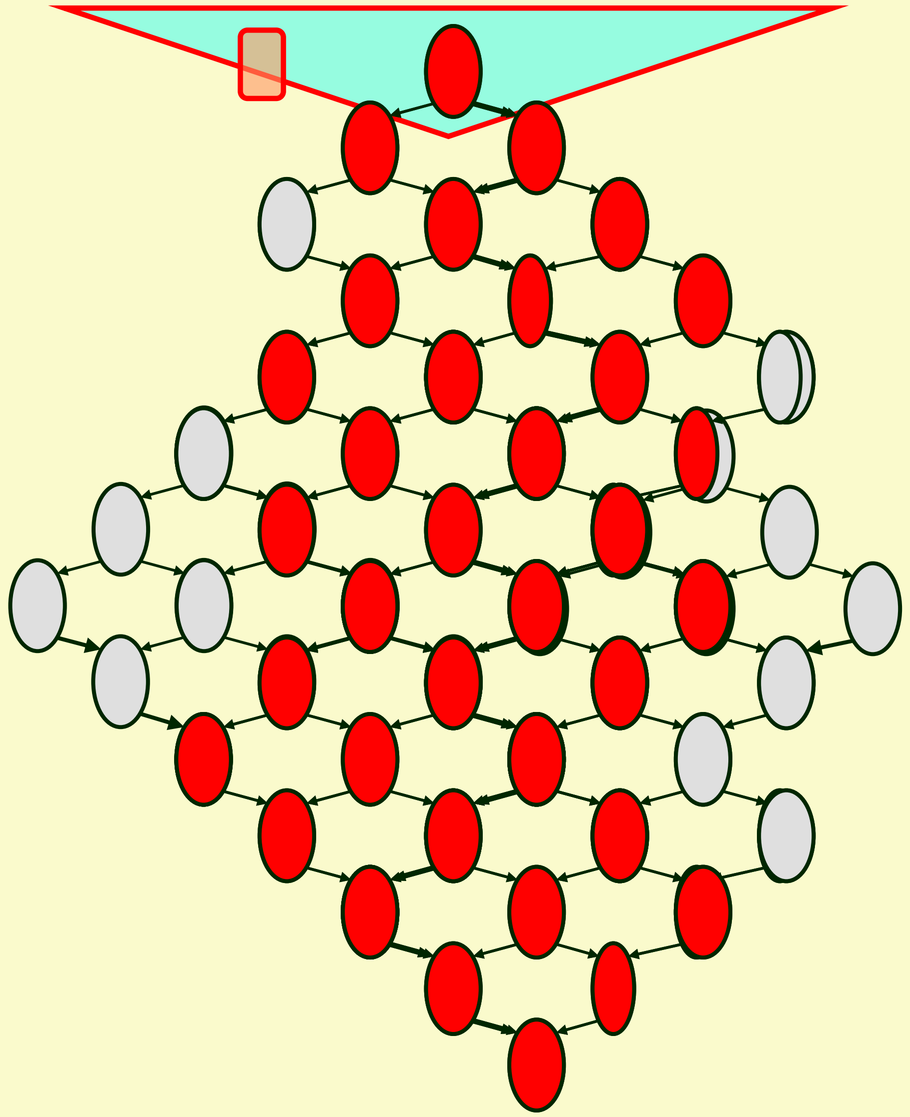
Predictive Monitoring

- Can predict the violation from the run that did not have the violation.
- Cannot detect a violation if there is no direct communication of intermediate value from p1 to p2

DiAna Architecture

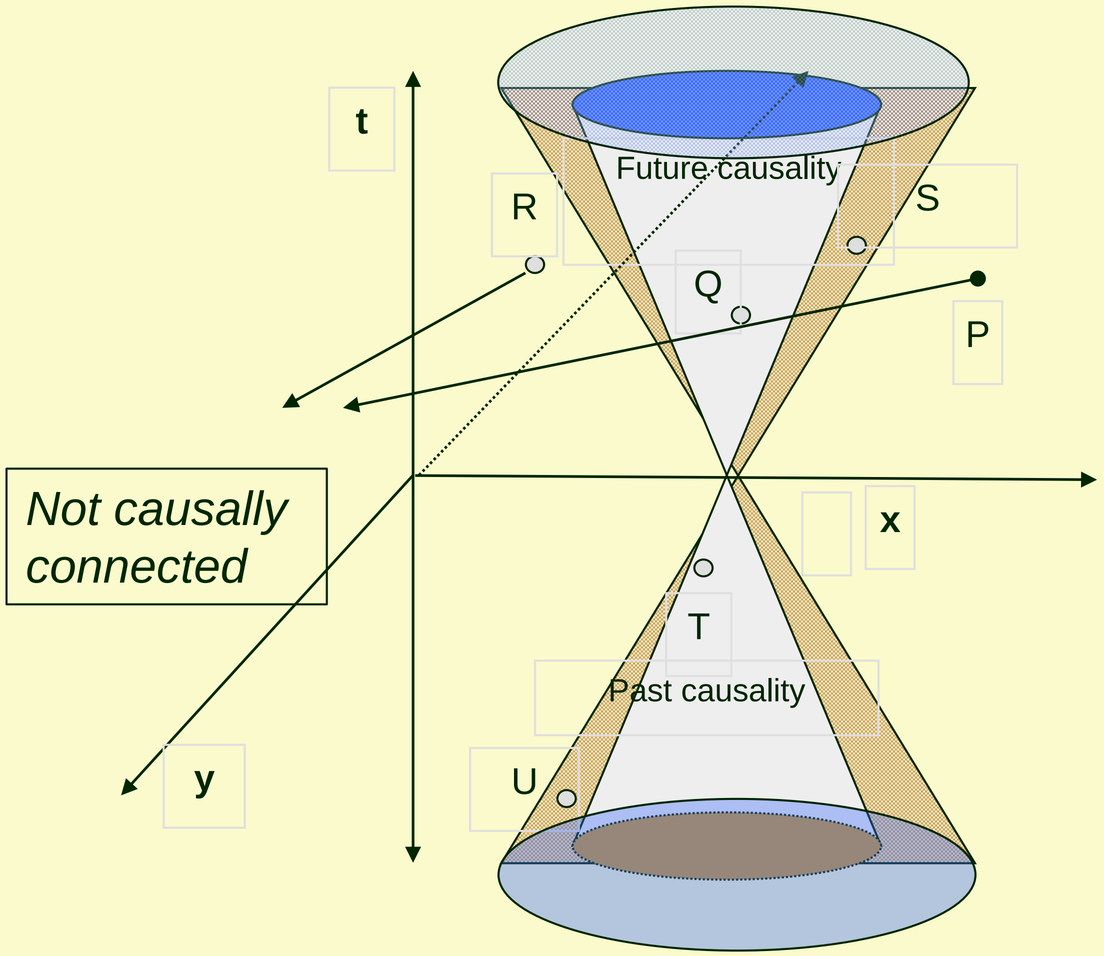


Causality Cone Heuristics



Probabilistic Programs

- An actor program is a probabilistic program in a distributed space with concurrent time.
- The behavior of a program is statistical in nature.

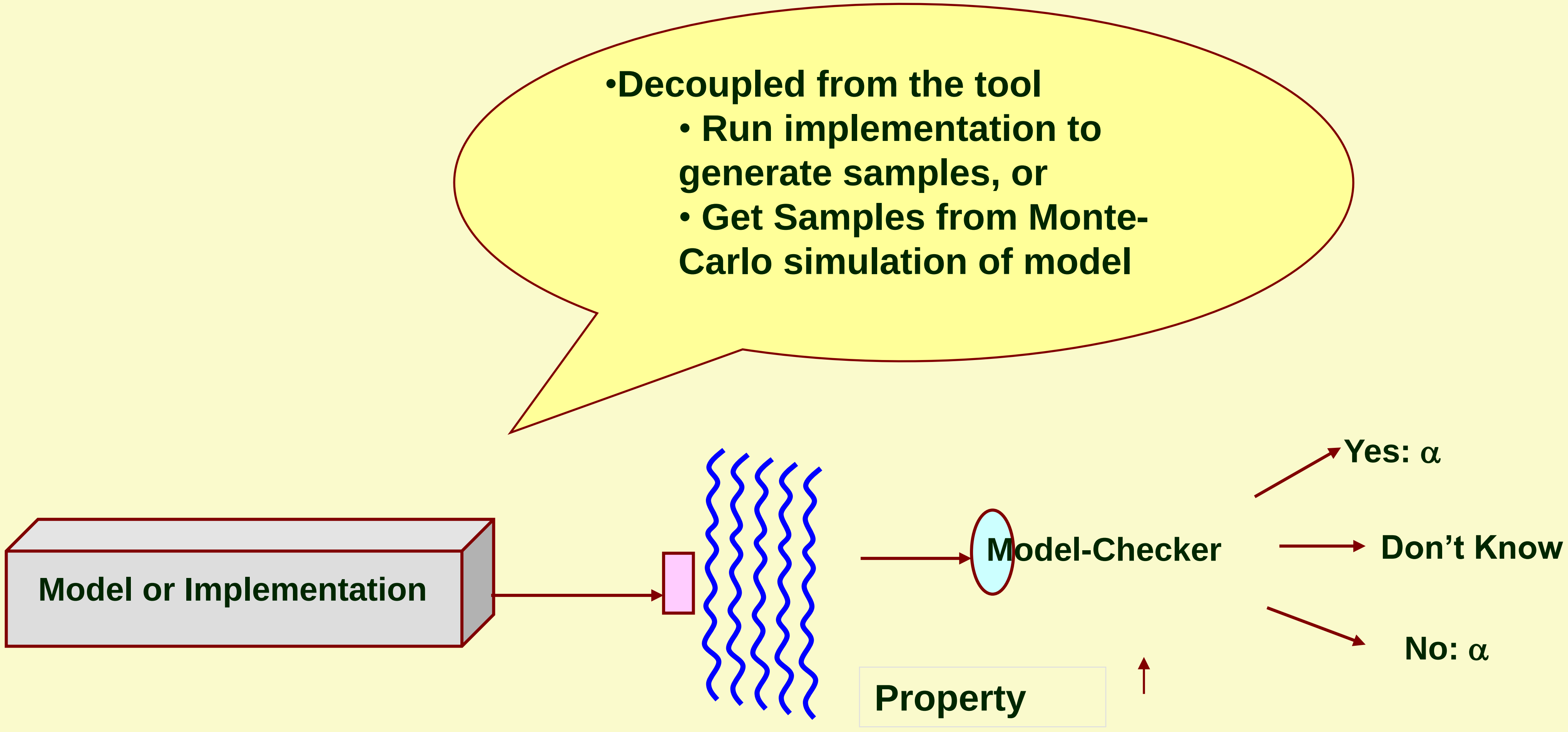


Properties in CSL sub-logic

- $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid : \phi \mid P_{\mathbf{Q}\ p}(\psi)$
- $\psi ::= \phi\ U^{<t}\ \phi \mid X\ \phi$

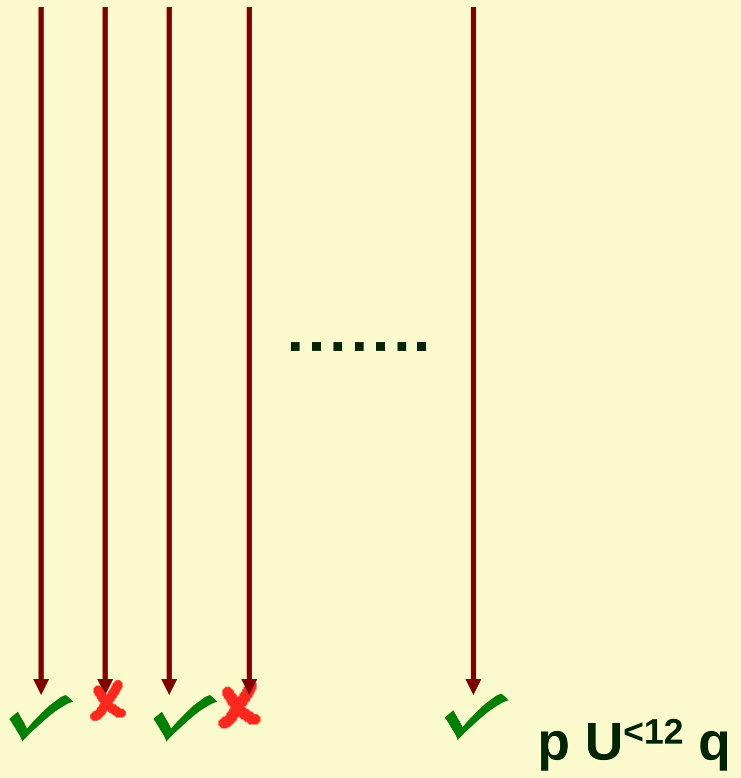
  where **Q 2** {<,>,¸,·}
- $P_{< 0.5}(\Diamond^{<10}\ \text{full})$
  - Probability that queue becomes full in 10 units of time is less than 0.5
- $P_{>0.98}(:\ \text{retransmit}\ U^{<200}\ \text{receive})$
  - Probability that a message is received successfully within 200 time units without any need for retransmission is greater than 0.98

Statistical Model Checking


•Decoupled from the tool
• Run implementation to generate samples, or
• Get Samples from Monte-Carlo simulation of model
Model or Implementation
Model-Checker
Property
Yes: α
Don’t Know
No: α

Checking $P_{<0.6}(p\ U^{<12}\ q)$ statistically at s

Sample contains, say, 30 paths from s



- On 21 paths $(p\ U^{<12}\ q)$ is satisfied
- 21/30 > 0.6
  - can we say that $P_{<0.6}(p\ U^{<12}\ q)$ is violated at s ??
  - Statistically, yes, provided we quantify the error in our decision
- error = α

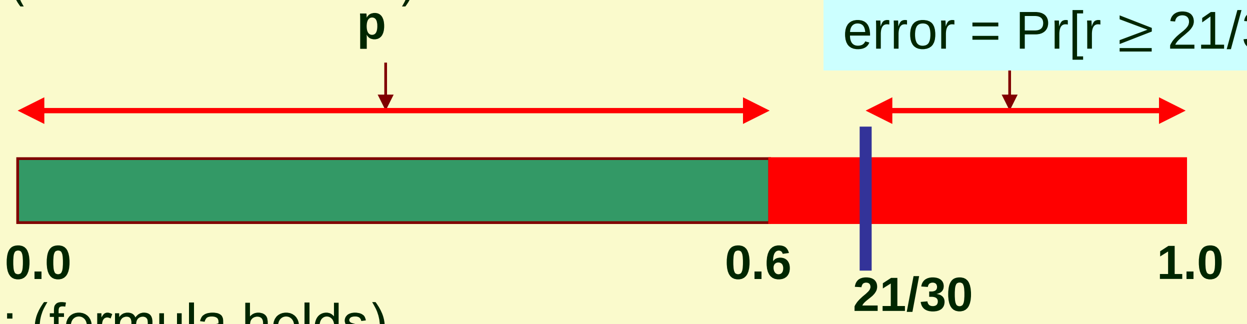
= Pr[On 21 (or more) out of 30 paths $(p\ U^{<12}\ q)$ hold | probability that $(p\ U^{<12}\ q)$ holds on a path is less than 0.6]

· Pr[X ¸ 21] where X~Binomial(30,0.6)

Error (p-value)

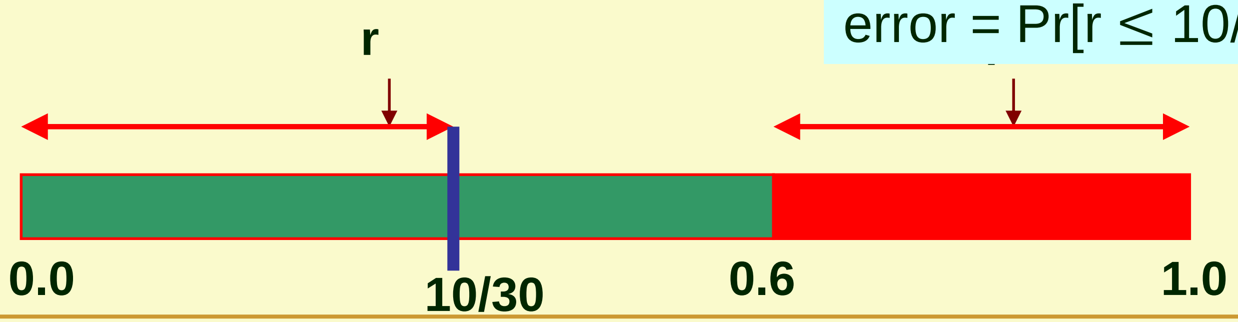
- Let r = (# of paths on which (p $U^{<12}$ q) hold / # of total paths)
- Let p = Pr[(p $U^{<12}$ q) holds on a path]
- "no" answer : (formula violates)

error = $\Pr[r \geq 21/30 \mid p \leq 0.6]$



- "yes" answer : (formula holds)

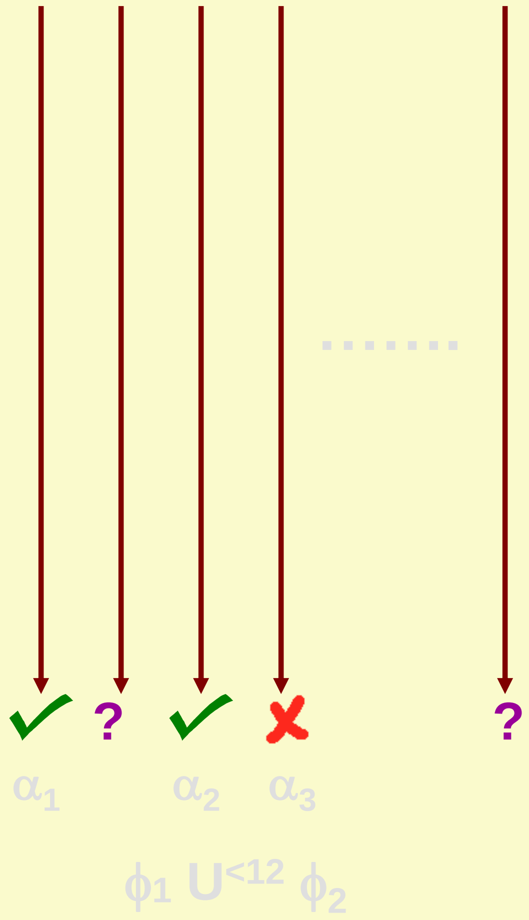
error = $\Pr[r \leq 10/30 \mid p > 0.6]$



Nested: Checking $P_{<0.6}(\phi_1 U^{<12}\phi_2)$ at s

- ϕ_1 and ϕ_2 contain nested probabilistic operators
- Checking $(\phi_1\ U^{<12}\ \phi_2)$ over a path
  - Answers are not simply "yes" or "no"
  - Answers can be
    - "yes" with error α
    - "no" with error α
    - "don't know"
- Need a modified decision procedure
  - Handle "don't know" to get useful answers
  - Incorporate error of decision for sub-formulas

Checking $P_{<0.6}(\phi_1 U^{<12} \phi_2)$ at s (Problem)

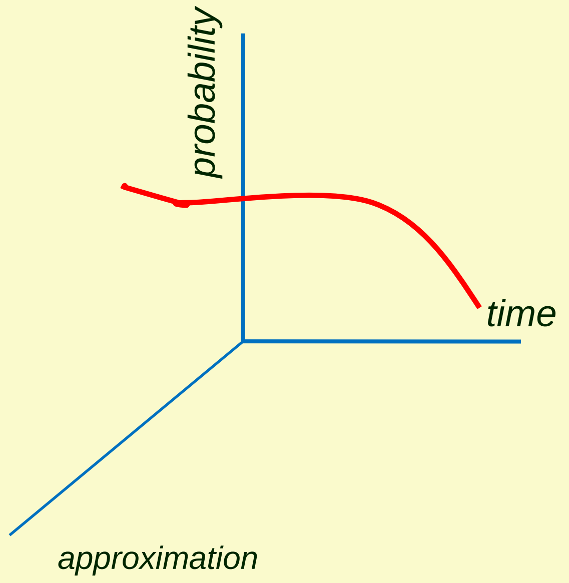


Solution

1. Resolve "don't know" (?) in adversarial fashion
   - Observation region
2. Create "uncertainty region" to incorporate error associated with sub-formulas.

Evolving Systems

- *Big data applications require approximate answers in a real-time.*
- *Probabilistic actor-based programming*
- *Adaptive programs:*
  - *Use predictive distributed monitoring* and *statistical inference.*
  - *Learning* and *prediction using Bayesian methods*



Representing State

- 100 nodes, 5 Abstract States → 5^{100} potential states
- Interested in aggregate properties or expected values
- Model state as pmf vector (superposition of probabilities)

$$s = \begin{array}{|c|c|c|c|c|}\hline s_1 & s_2 & s_3 & \cdots & s_n \\ \hline\end{array}$$

$$s = p_1 s_1 + p_2 s_2 + \ldots + p_n s_n$$

Evolution of Probability Distributions

- Transitions may be governed by a Markov model
- pmf vector defines the initial state for a DTMC
  - Search in an *Euclidean space*
  - Property stabilizes after a computable depth
- Model checking reduced to linear algebra
- *Euclidean Model Checking*

Conclusions

- Programming based on the Actor Model facilitates scalable, secure development of concurrent programs.
  - Probabilistic programming methods needed
- New reasoning methods needed:
  - Scalable
  - Model probabilistic computation
  - Address quantitative properties

References I: Actors

- Gul Agha, *ACTORS - a model of concurrent computation in distributed systems*, MIT Press Series in Artificial Intelligence. MIT Press, 1986.
- Gul Agha, “Concurrent object-oriented programming,” *Communications of the ACM*, 33(9):125–141, Sep 1990.
- Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott, “Towards a theory of actor computation,” in Rance Cleaveland, editor, CONCUR ’92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings, volume 630 of Lecture Notes in Computer Science, pages 565–579. Springer, 1992.
- Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott, “A foundation for actor computation,” Journal of Functional Programming, 7(1):1–72, 1997.
- WooYoung Kim and Gul Agha. “Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages”. In: Proceedings of Supercomputing ’95, San Diego, CA, USA, December 4-8, 1995. Ed. by Sidney Karin. IEEE Computer Society / ACM, 1995, 39pp.

References II: Actor Languages and Coordination

- Christopher R. Houck and Gul Agha. “HAL: A High-Level Actor Language and Its Distributed Implementation”. In: Proceedings of the 1992 International Conference on Parallel Processing (ICPP). 1992, pp. 158–165. Svend Frolund and Gul Agha. “A Language Framework for Multi-Object Coordination”. In: *ECOOP’93 - Object-Oriented Programming, 7th European Conference*, Kaiserslautern, Germany, July 26-30, 1993, Proceedings. Ed. by Oscar Nierstrasz. Vol. 707. Lecture Notes in Computer Science. Springer, 1993, pp. 346–360.
- Gul Agha, Svend Frolund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel C. Sturman. “Abstraction and modularity mechanisms for concurrent computing”. In: *IEEE Parallel and Distributed Technology* 1.2 (1993), pp. 3–14.
- Carlos A. Varela and Gul Agha. “Programming Dynamically Reconfigurable Open Systems with SALSA”. In: SIGPLAN Notices. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications: Onward! Track. Vol. 36. 12. 2001, pp. 20–34
- Peter Dinges and Gul Agha. “Scoped Synchronization Constraints for Large Scale Actor Systems”. In: *Coordination Models and Languages - 14th International Conference*, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings. Ed. by Marjan Sirjani. Vol. 7274. Lecture Notes in Computer Science. Springer, 2012, pp. 89–103.

References III: Concolic Testing

- K. Sen, D. Marinov, and G. Agha, “Cute: a concolic unit testing engine for C,” In M. Wermelinger and H. Gall, editors, *Proceedings of the 10th European Software Engineering Conference* held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005, pages 263–272. ACM, 2005.
- K. Sen and G. Agha, “A race-detection and flipping algorithm for automated testing of multi-threaded programs,” In E. Bin, A. Ziv, and S. Ur, editors, *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference*, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers, volume 4383 of Lecture Notes in Computer Science, pages 166–182. Springer, 2007.
- P. Dinges and G. Agha “Solving complex path conditions through heuristic search on induced polytopes,” In Proceedings of the 22nd ACM SIGSOFT *International Symposium on Foundations of Software Engineering*, FSE 2014, pages 425–436. ACM, 2014.
- Peter Dinges and Gul Agha. 2014. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (ASE '14). ACM, New York, NY, USA, 31-36. DOI=http://dx.doi.org/10.1145/2642937.2642951

References IV: Actor Testing Tools

- K. Sen and G. Agha, “Automated systematic testing of open distributed programs,” In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference,* FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings, volume 3922 of Lecture Notes in Computer Science, pages 339–356. Springer, 2006.
- Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. 2009. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (ASE '09). IEEE Computer Society, Washington, DC, USA, 468-479. DOI=http://dx.doi.org/10.1109/ASE.2009.88
- Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. 2010. Basset: a tool for systematic testing of actor programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (FSE '10). ACM, New York, NY, USA, 363-364. DOI=http://dx.doi.org/10.1145/1882291.1882349

References V: Runtime Verification

- Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2004. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceedings of the 26th International Conference on Software Engineering* (ICSE '04). IEEE Computer Society, Washington, DC, USA, 418-427.
- Sen, Koushik, Grigore Roşu, and Gul Agha. "Online efficient predictive safety analysis of multithreaded programs." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2004.

References VI: Probabilistic Programming, Statistical Model Checking and Euclidean Model Checking

- Kwon, YoungMin, and Gul Agha. "Linear inequality LTL (iLTL): A model checker for discrete time markov chains." *International conference on formal engineering methods*. Springer Berlin Heidelberg, 2004.
- Sen, Koushik, Mahesh Viswanathan, and Gul Agha. "Statistical model checking of black-box probabilistic systems." *International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2004.
- Sen, Koushik, Mahesh Viswanathan, and Gul Agha. "On statistical model checking of stochastic systems." *International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2005.
- Agha, Gul, José Meseguer, and Koushik Sen. "PMaude: Rewrite-based specification language for probabilistic object systems." *Electronic Notes in Theoretical Computer Science* 153.2 (2006): 213-239.
- Korthikanti, Vijay Anand, et al. "Reasoning about MDPs as transformers of probability distributions." *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*. IEEE, 2010.
- Kwon, YoungMin, and Gul Agha. "Verifying the evolution of probability distributions governed by a DTMC." *IEEE Transactions on Software Engineering* 37.1 (2011): 126-141.

Sensor Network Application

- Rice, Jennifer A., et al. "Flexible smart sensor framework for autonomous structural health monitoring." *Smart structures and Systems* 6.5-6 (2010): 423-438.
- Jang, Shinae, et al. "Structural health monitoring of a cable-stayed bridge using smart sensor technology: deployment and evaluation." *Smart Structures and Systems* 6.5-6 (2010): 439-459.
- Kwon, YoungMin, Kirill Mechitov, and Gul Agha. "Design and implementation of a mobile actor platform for wireless sensor networks." *Concurrent objects and beyond*. Springer Berlin Heidelberg, 2014. 276-316.
- Khamespanah, Ehsan, Kirill Mechitov, Marjan Sirjani, and Gul Agha. "Schedulability Analysis of Distributed Real-Time Sensor Network Applications Using Actor-Based Model Checking." In *International Symposium on Model Checking Software*, pp. 165-181. Springer International Publishing, 2016.