Model Checking of Threshold-based Fault-tolerant Distributed Algorithms

and beyond ...

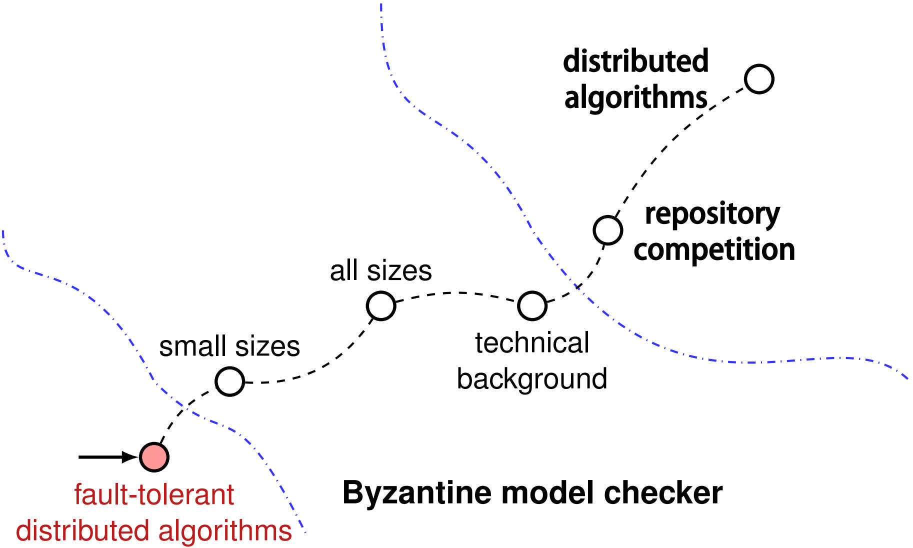
Helmut Veith

joint work with

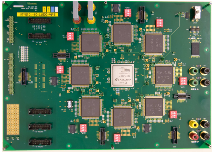
Annu Gmeiner   Igor Konnov   Ulrich Schmid   Josef Widder

TU WIEN   forsyte Formal Methods in Systems Engineering   Embedded Computing Systems   RiSE Rigorous Systems Engineering

Our journey


distributed
algorithms
repository
competition
all sizes
small sizes
technical
background
fault-tolerant
distributed algorithms
Byzantine model checker

Distributed Systems



Are they always working?

No. . . some failing systems

Ariane 501 maiden flight (1996)

primary/backup, i.e., 2 replicated computers

both run into the same overflow

Qantas Airbus in-flight Learmonth upset (2008)

1 out of 3 replicated components failed

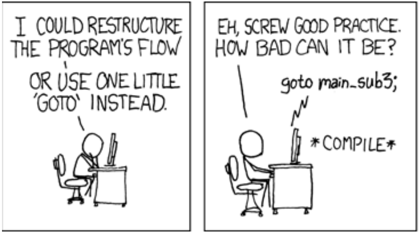
computer initiated dangerous altitude drop

Why do they fail?

1. **Design & implementation bugs**

   approach: find the bugs and fix them

   tools: model checking, static analysis



[xkcd.com/292]

2. **Runtime faults**

   outside of control of designer/developer

   approach: replicate & coordinate

   tools: fault-tolerant distributed algorithms

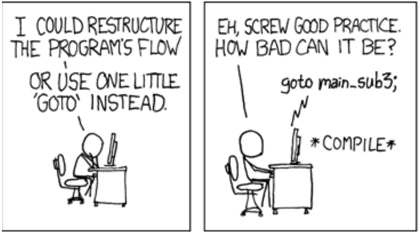
are they always working?

Why do they fail?

1. **Design & implementation bugs**

   approach: find the bugs and fix them

   tools: model checking, static analysis



[xkcd.com/292]

2. **Runtime faults**

   outside of control of designer/developer

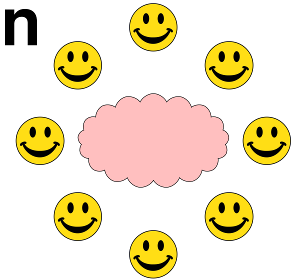
   approach: replicate & coordinate

   tools: fault-tolerant distributed algorithms

are they always working?

Driscoll (Honeywell)

Fault-tolerant distributed algorithms

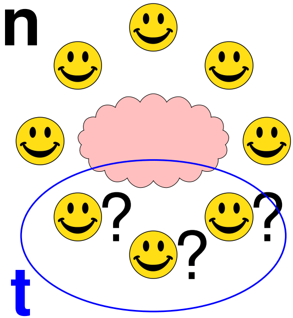


n processes communicate by sending messages

all processes know that at most t of them might be faulty

f are actually faulty (and $n > 3t \wedge t \geq f \geq 0$)

Fault-tolerant distributed algorithms

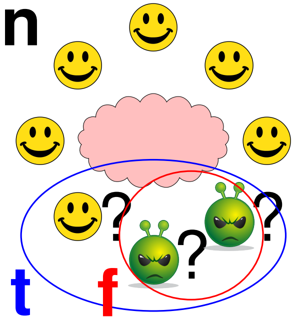


n processes communicate by sending messages

all processes know that at most t of them might be faulty

f are actually faulty (and $n > 3t \wedge t \geq f \geq 0$)

Fault-tolerant distributed algorithms



n processes communicate by sending messages

all processes know that at most *t* of them might be faulty

f are actually faulty (and $n > 3t \wedge t \geq f \geq 0$)

Reliable Broadcast by Srikanth & Toueg 87

```
if initiator then send INIT to all;

while true do
 if received INIT from at least 1 distinct processes
 then send ECHO to all;

 if received ECHO from at least t + 1 distinct processes
    and not sent ECHO before
 then send ECHO to all;

 if received ECHO from at least n - t distinct processes
 then accept;
od
```

It works correctly when:

out of $n > 3t$ processes, $f \leq t$ processes are faulty (Byzantine)

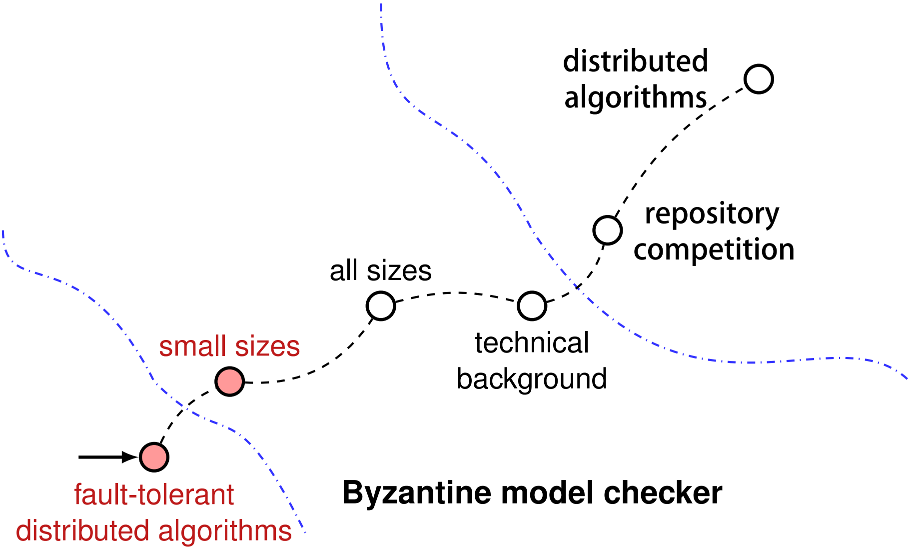
Reliable broadcast: properties

Unforgeability: If no correct process receives “broadcast”, then no correct process ever accepts.

Correctness: If all correct processes receive “broadcast”, then at least one correct process accepts.

Relay: Whenever a correct process accepts, eventually all correct processes accept.

Our journey


distributed
algorithms
repository
competition
all sizes
technical
background
small sizes
fault-tolerant
distributed algorithms
Byzantine model checker

What do we want to verify?

The algorithms come in pseudo code and English:

is it ok to assign Byzantine processes right in the initial state?

yes, it is folklore knowledge

We chose PROMELA as a modeling language:

- we can use SPIN
- model checking community knows it
- it does not shock people from distributed algorithms
- Promela forces us to do a lot of hacking

What do we want to verify?

The algorithms come in pseudo code and English:

is it ok to assign Byzantine processes right in the initial state?

yes, it is folklore knowledge

We chose PROMELA as a modeling language:

- we can use SPIN
- model checking community knows it
- it does not shock people from distributed algorithms

Promela forces us to do a lot of hacking

Encoding reliable broadcast in Promela

Parametric Promela code:

```
int nsnt = 0;
active[n-f] proctype P() {
 byte pc, nrcvd;
 byte npc, nnrcvd;
 ...
 if
 :: nrcvd + 1 < nsnt + f
    -> nrcvd++;
 :: skip;
 fi;
 if
 :: nnrcvd >= n - t
    -> npc = ACCEPT;
 :: nnrcvd < n - t
   && nnrcvd >= t + 1
   -> npc = SENT; nsnt++;
 ...
 fi;
```

Similar TLA$^+$ code:

CONSTANTS $n,\ t,\ f$

VARIABLE $pc,\ rcvd,\ sent$

$$vars \triangleq \langle pc,\ rcvd,\ sent \rangle$$

$$\begin{aligned}
&Receive(self) \triangleq \\
&\quad \exists\, r \in \text{SUBSET}\ (P \times \{\text{"ECHO"}\}) : \\
&\qquad \land\ r \subseteq sent \cup \{\langle p, \text{"ECHO"}\rangle : p \in Faulty\} \\
&\qquad \land\ rcvd[self] \subseteq r \\
&\qquad \land\ rcvd' = [rcvd\ \text{EXCEPT}\ ![self] = r]
\end{aligned}$$

$$\begin{aligned}
&UponNonFaulty(self) \triangleq \\
&\quad \land\ pc[self] \neq \text{"SENT"} \\
&\quad \land\ Cardinality(rcvd'[self]) \geq t + 1 \\
&\quad \land\ Cardinality(rcvd'[self]) < n - t \\
&\quad \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{"SENT"}] \\
&\quad \land\ sent' = sent \cup \{\langle self, \text{"ECHO"}\rangle\}
\end{aligned}$$

$$\begin{aligned}
&UponAccept(self) \triangleq \\
&\quad \land\ pc[self] = \text{"SENT"} \\
&\quad \land\ Cardinality(rcvd'[self]) \geq n - t \\
&\quad \land\ pc' = [pc\ \text{EXCEPT}\ ![self] = \text{"ACCEPT"}] \\
&\quad \land\ sent' = sent
\end{aligned}$$

...

Checking small instances

We consider a number of threshold-based algorithms.

1. Reliable broadcast for Byzantine faults (BYZ)
2. Reliable broadcast for omission faults (OMIT)
3. Reliable broadcast for symmetric faults (SYMM)
4. Reliable broadcast for clean crashes (CLEAN)

[Srikanth & Toueg 87, **STRB**]

5. Folklore reliable broadcast for clean crashes

[Chandra & Toueg 96, **FRB**]

6. Asynchronous Byzantine agreement

[Bracha & Toueg 85, **ABA**]

7. Condition-based consensus (crash faults)

[Mostéfaoui et al. 01, **CBC**]

Reference: other algorithms used later

9. Non-blocking atomic commit

[Raynal 97, **NBAC**]

10. Non-blocking atomic commit with failure detectors

[Guerraoui 01, **NBACG**]

11. Folklore one-step consensus

[Dobre, Suri 06, **CF1S**]

12. Consensus in one communication step

[Brasileiro 01, **C1CS**]

13. BOSCO: One-step Byzantine Asynchronous Consensus

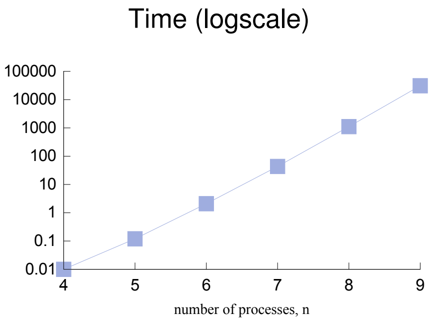
[Song, von Renesse 08, **BOSCO**]

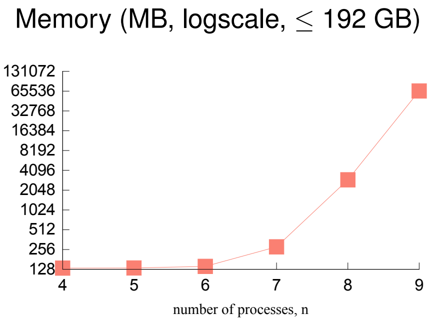
Experiments with small instances

Algorithm	Fault	Parameters	Resilience	Properties	Time
1. STRB	Byz	$n = 7, t = 2, f = 2$	$n > 3t$	**U**, **C**, **R**	6 sec.
1. STRB	Byz	$n = 7, t = 3, f = 2$	$n > 3t$	**U**, **C**, **R**	5 sec.
1. STRB	Byz	$n = 7, t = 1, f = 2$	$n > 3t$	**U**, **C**, **R**	1 sec.
2. STRB	Omit	$n = 5, t = 2, f = 2$	$n > 2t$	**U**, **C**, **R**	4 sec.
2. STRB	Omit	$n = 5, t = 2, f = 3$	$n > 2t$	**U**, **C**, **R**	5 sec.
3. STRB	Symm	$n = 5, t = 1, f_p = 1, f_s = 0$	$n > 2t$	**U**, **C**, **R**	1 sec.
3. STRB	Symm	$n = 5, t = 2, f_p = 3, f_s = 1$	$n > 2t$	**U**, **C**, **R**	1 sec.
4. STRB	Clean	$n = 3, t = 2, f_c = 2, f_{nc} = 0$	$n > t$	**U**, **C**, **R**	1 sec.
5. FRB	Crash	$n = 2$	—	**U**, **C**, **R**	1 sec.
6. ABA	Byz	$n = 5, t = 1, f = 1$	$n > 3t$	**R**	131 sec.
6. ABA	Byz	$n = 5, t = 1, f = 2$	$n > 3t$	**R**	1 sec.
6. ABA	Byz	$n = 5, t = 2, f = 2$	$n > 3t$	**R**	1 sec.
7. CBC	Crash	$n = 3, t = 1, f = 1$	$n > 2t$	**V0**, **V1**, **A**, **T**	1 sec.
7. CBC	Crash	$n = 3, t = 1, f = 2$	$n > 2t$	**V0**, **V1**, **A**, **T**	1 sec.

Adding more processes

Checking reliable broadcast with one Byzantine fault in Spin:

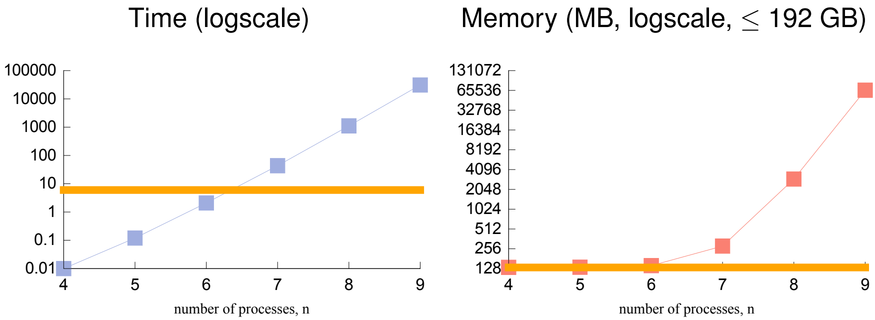




Can general-purpose model checkers scale up to 1000 processes?

We focus on fault-tolerant distributed algorithms

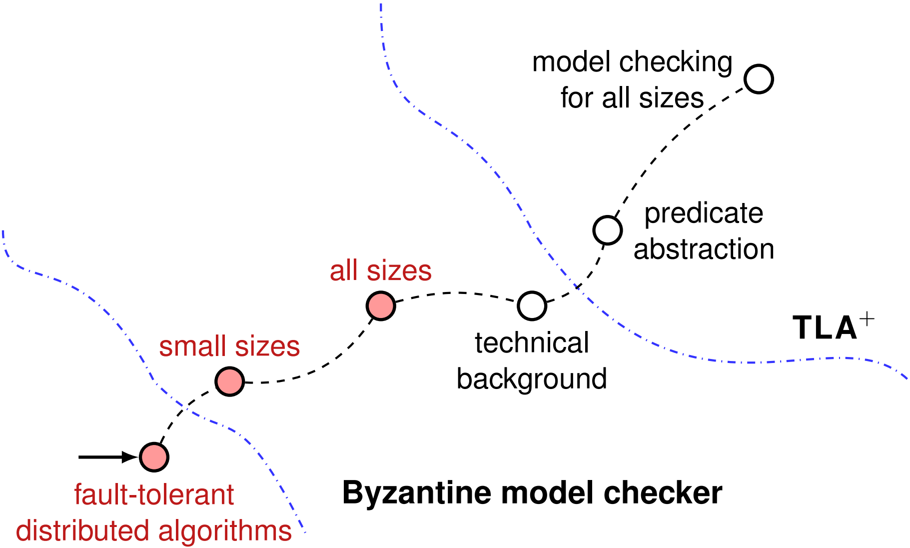
Checking for all sizes



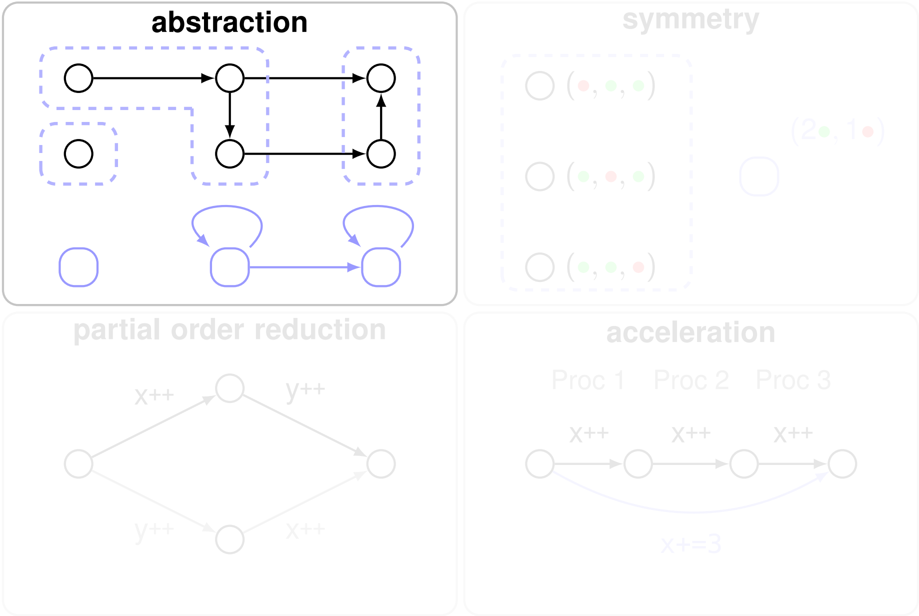
Checking once and for all sizes faster

than checking a system of 7 processes

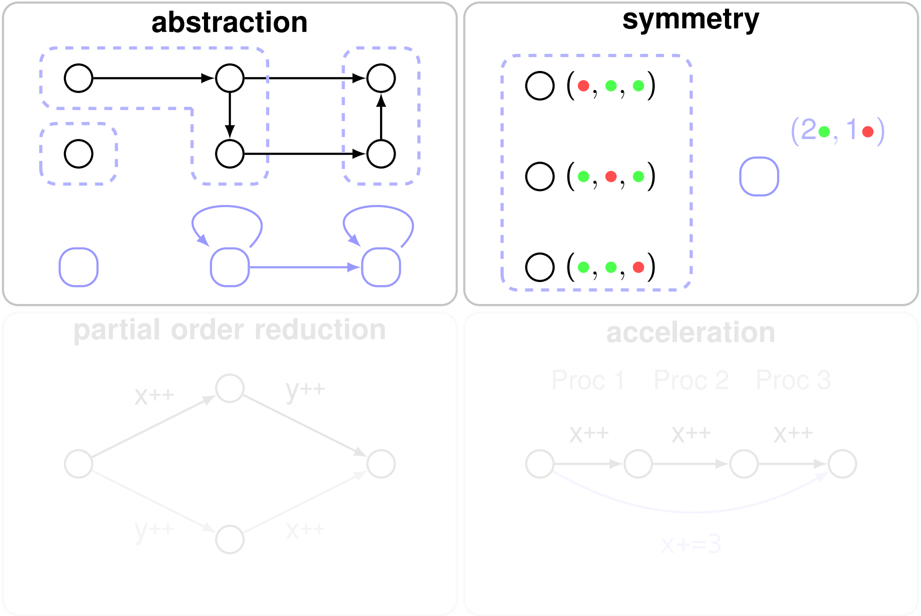
Our journey


model checking
for all sizes
predicate
abstraction
all sizes
technical
background
TLA+
small sizes
fault-tolerant
distributed algorithms
Byzantine model checker

Our mathematical tools

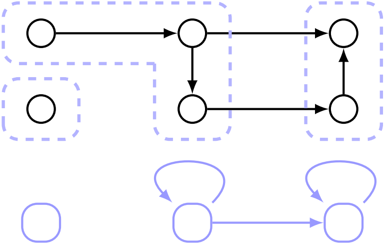


Our mathematical tools

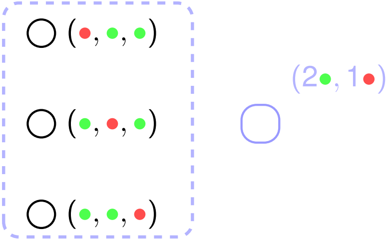


Our mathematical tools

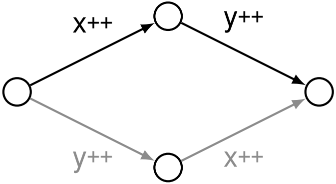
abstraction



symmetry



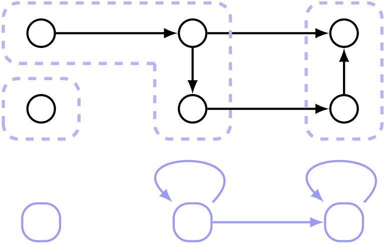
partial order reduction



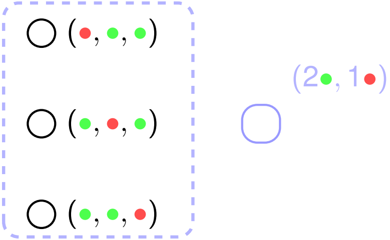
acceleration

Our mathematical tools

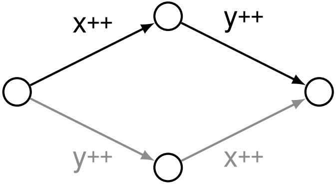
abstraction



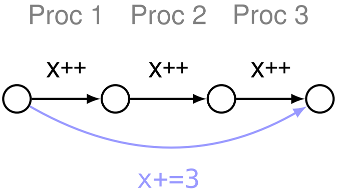
symmetry



partial order reduction



acceleration



Stacks of techniques

FMCAD'13	CONCUR'14	CAV'15
	partial orders & acceleration	partial orders & acceleration
counter abstraction	counter abstraction	counters in SMT
symmetry	symmetry	symmetry
data abstraction	data abstraction	data abstraction
state enumeration or BDDs	bounded model checking	bounded model checking
SPIN, NuSMV-BDD	**NuSMV-SAT**	**SMT**

Our benchmarks

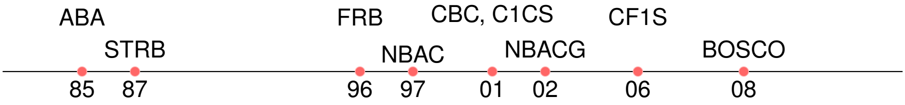
Now we can verify safety of the parameterized algorithms:

Reliable broadcast (FRB, STRB, ABA)

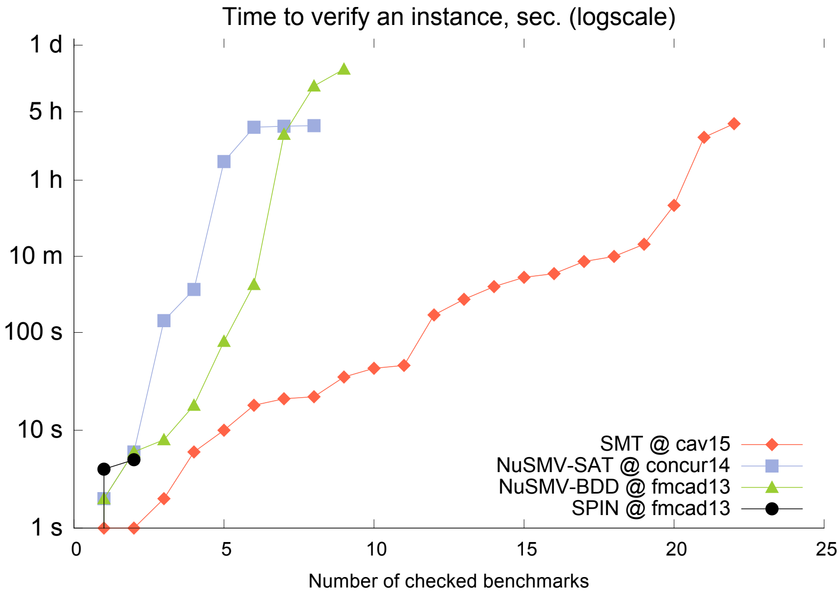
Non-blocking atomic commit with failure detectors (NBAC, NBACG)

Condition-based consensus (CBC)

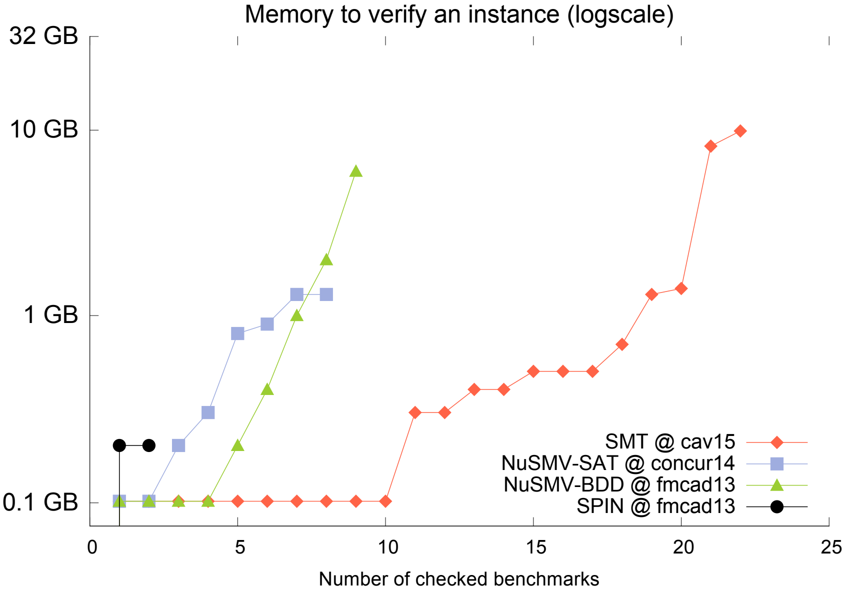
One-step consensus (CF1S, C1CS, BOSCO)



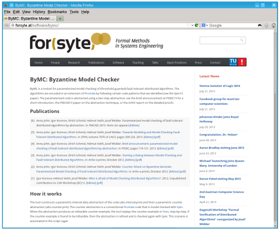
Our recent breakthroughs (time)

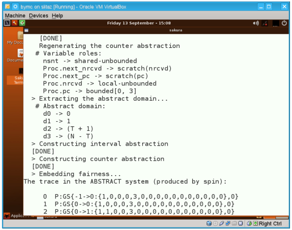


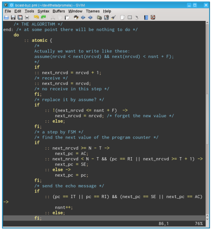
Our recent breakthroughs (memory)



Byzantine model checker







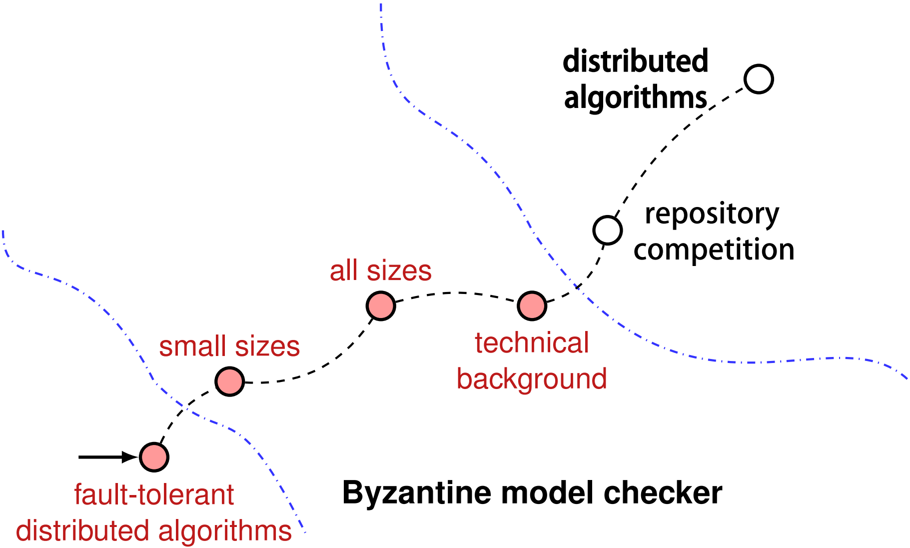
A virtual machine with full setup:

the tool in OCaml

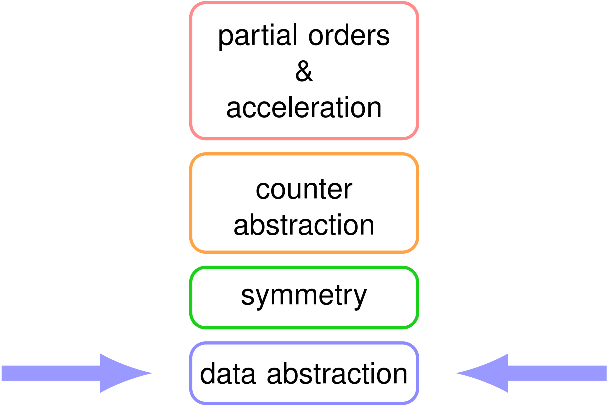
our benchmarks in parametric Promela

[http://forsyte.at/software/bymc]

Our journey


distributed
algorithms
repository
competition
all sizes
small sizes
technical
background
fault-tolerant
distributed algorithms
Byzantine model checker

Position in the stack



Data abstraction

Concrete values are not important

Thresholds are essential:

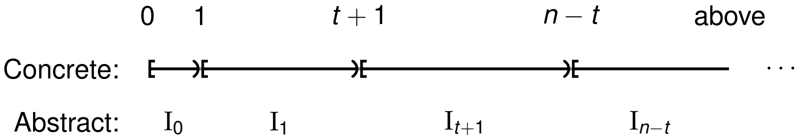
$$0, 1, t + 1, n - t$$

Intervals with symbolic boundaries:

- $\mathrm{I}_0 = [0, 1)$
- $\mathrm{I}_1 = [1, t + 1)$
- $\mathrm{I}_{t+1} = [t + 1, n - t)$
- $\mathrm{I}_{n-t} = [n - t, \infty)$

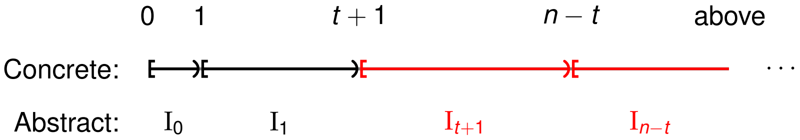
```
int nsnt = 0;
active[n-f] proctype P() {
 byte pc, nrcvd;
 byte npc, nnrcvd;
 ...
 if
 :: nrcvd + 1 < nsnt + f
    -> nrcvd++;
 :: skip;
 fi;
 if
 :: nnrcvd >= n - t
    -> npc = ACCEPT;
 :: nnrcvd < n - t
   && nnrcvd >= t + 1
   -> npc = SENT; nsnt++;
 ...
 fi;
```

Abstract operations on message counters



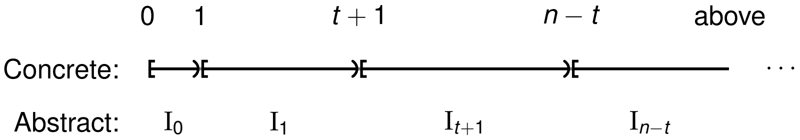
Concrete $t + 1 \leq x$

Abstract operations on message counters



Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

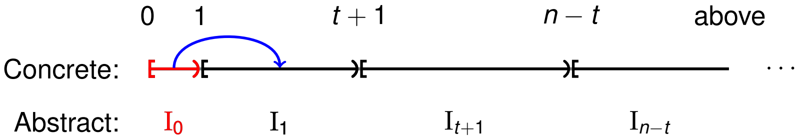
Abstract operations on message counters



Concrete $t + 1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

Concrete $x' = x + 1$,

Abstract operations on message counters

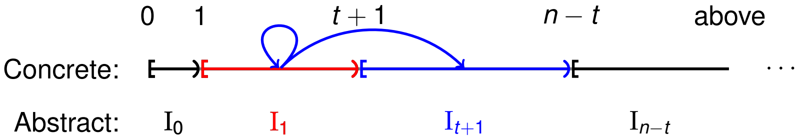


Concrete $t + 1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

$$x = \mathrm{I}_0 \quad \wedge \; x' = \mathrm{I}_1 \; \ldots$$

Abstract operations on message counters

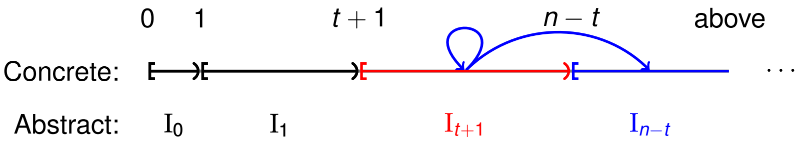


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

Concrete $x' = x + 1,$ is abstracted as:

$$\begin{aligned} & x = \mathrm{I}_0 \quad \wedge \; x' = \mathrm{I}_1 \\ \vee & x = \mathrm{I}_1 \quad \wedge (x' = \mathrm{I}_1 \quad \vee x' = \mathrm{I}_{t+1}) \ldots \end{aligned}$$

Abstract operations on message counters

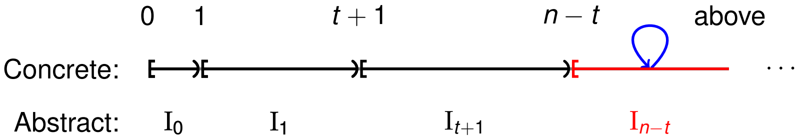


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

$$
\begin{aligned}
& x = \mathrm{I}_0 && \wedge\ x' = \mathrm{I}_1 \\
\vee & x = \mathrm{I}_1 && \wedge\ (x' = \mathrm{I}_1 \quad \vee\ x' = \mathrm{I}_{t+1}) \\
\vee & x = \mathrm{I}_{t+1} && \wedge\ (x' = \mathrm{I}_{t+1} \vee x' = \mathrm{I}_{n-t}) \ \ldots
\end{aligned}
$$

Abstract operations on message counters

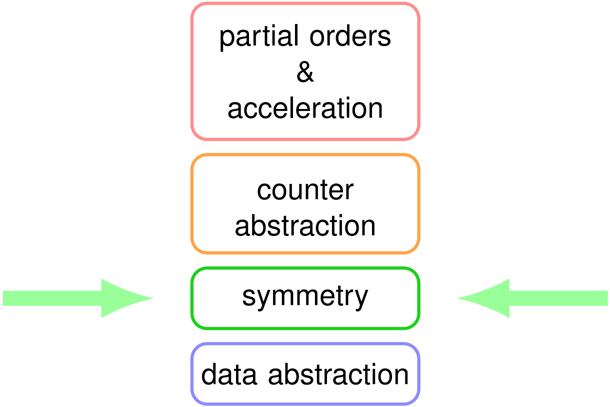


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

$$
\begin{aligned}
& x = \mathrm{I}_0 \quad \wedge \ x' = \mathrm{I}_1 \\
\vee & x = \mathrm{I}_1 \quad \wedge (x' = \mathrm{I}_1 \quad \vee x' = \mathrm{I}_{t+1}) \\
\vee & x = \mathrm{I}_{t+1} \wedge (x' = \mathrm{I}_{t+1} \vee x' = \mathrm{I}_{n-t}) \\
\vee & x = \mathrm{I}_{n-t} \wedge \ x' = \mathrm{I}_{n-t}
\end{aligned}
$$

Position in the stack

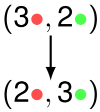


Symmetry and counter representation

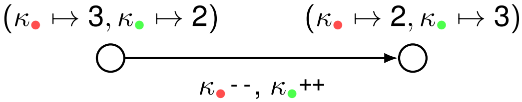
Our benchmarks do not use process ids

These transitions are indistinguishable:

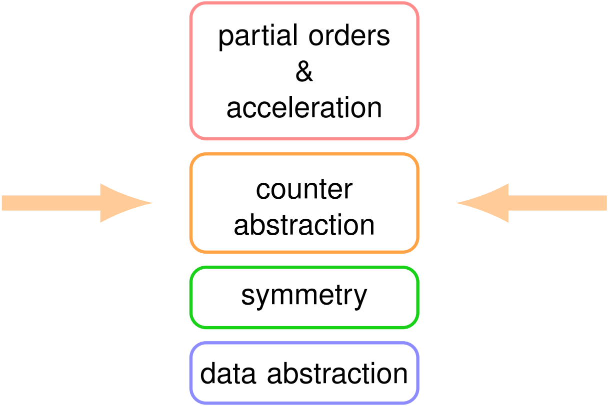
We just count processes in different states:



OR



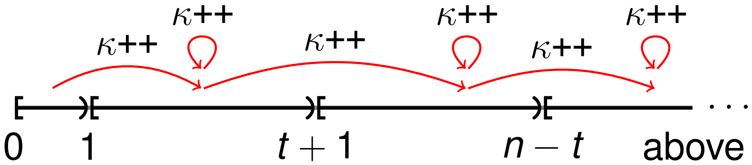
Position in the stack



Counter abstraction

Abstract counters over the intervals,

e.g., $\{[0,1), [1, t+1), [t+1, n-t), [n-t, \infty)\}$



A global state looks like $(\kappa_{\bullet} \mapsto \mathrm{I}_1, \kappa_{\bullet} \mapsto \mathrm{I}_{t+1})$

Soundness of the abstractions

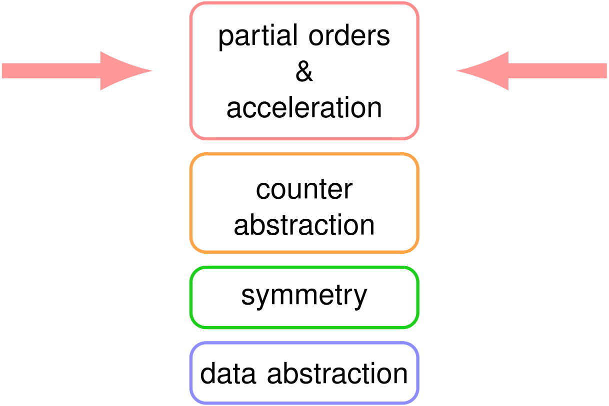
If the model checker tells us that there is no bug in the abstract model, then there is no bug for any system size.

This works both for safety and liveness.

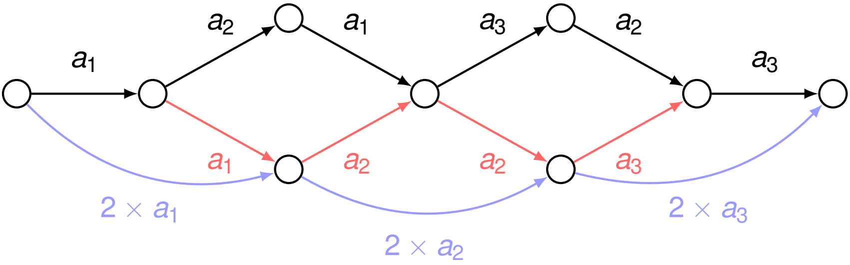
> Faulty processes cannot forge broadcast
>
> Correct processes eventually agree on broadcast

Formally proven in [FMCAD'13].

Position in the stack



Partial orders and acceleration



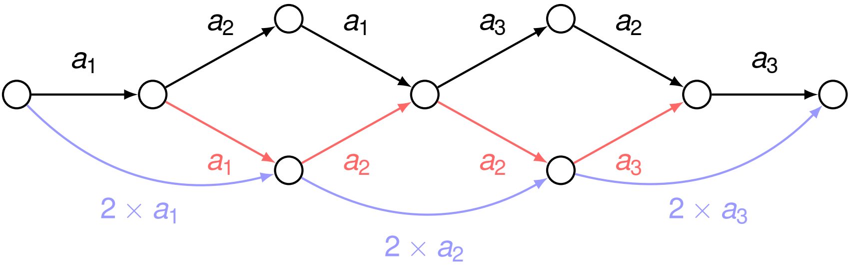
We can compute a bound on the diameter of the accelerated system

Theorem [CONCUR'14]

The bound depends only on the process code, not the parameter values

Result: safety bugs are always caught with bounded model checking

Bounded executions in SMT



fixed parameters: a representative (accelerated) execution

all parameters: a pattern to generate the representative executions

$$a_1^* \ a_2^* \ a_3^* \quad \text{captures} \quad a_1^2 \ a_2^2 \ a_3^2 \quad \text{and} \quad a_1^3 \ a_2^3 \ a_3^3$$

SMT solver checks, whether a pattern generates a bad execution

Z3, MathSAT, etc.

Complete parameterized reachability

Sound and complete algorithm for parameterized reachability

Let Φ be the set of all guards in the process code,

e.g., $\Phi = \{nsnt \geq t + 1, nsnt \geq n - t\}$

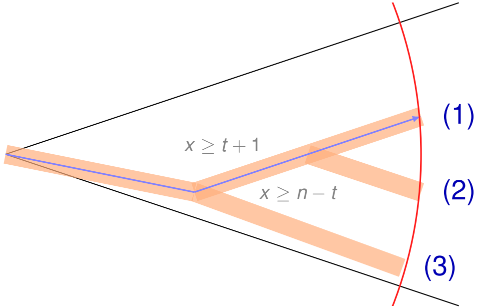
and $\mathcal{R}$ be the set of all process transitions

Theorem [CAV'15]

There is a set of at most $|\Phi|!$ patterns generating all representative executions

Each pattern is no longer than $(3 \cdot |\Phi| + 2) \cdot |\mathcal{R}|$

Distributed reachability checking?



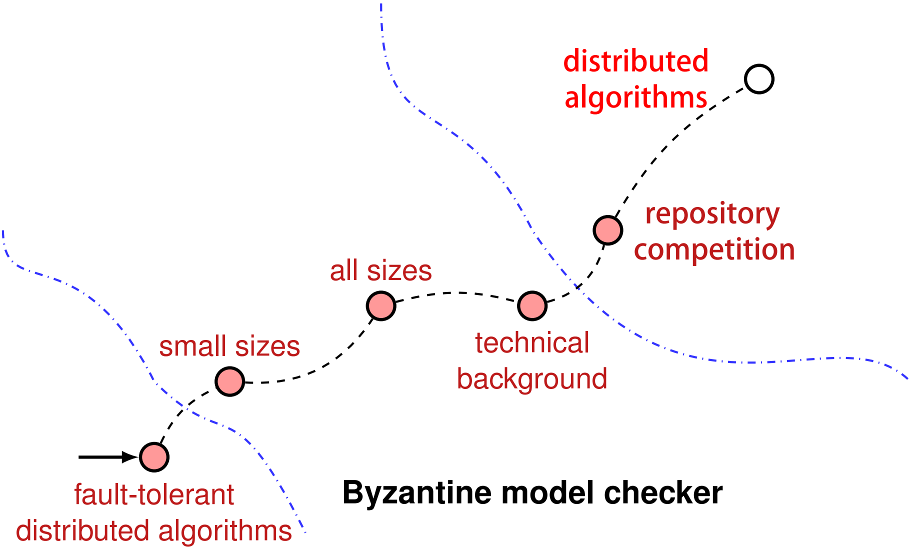
We enumerate patterns and check them in SMT solvers:

they can be tried independently, on different machines

We have not tried it yet

Checking Paxos in the cloud?

Our journey



Byzantine model checker

DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

SINCE 2011, ENGINEERS at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experience we refer to the authors by their initials.

At AWS we strive to build services that are simple for customers to use. External simplicity is built on a hidden substrate of complex distributed systems. Such complex internals are required to achieve high availability while running on cost-efficient infrastructure and cope with relentless business growth. As an example of this growth, in 2006, AWS launched S3, its Simple Storage Service. In the following six years, S3 grew to store one

S3 is just one of many AWS services that store and process data our customers have entrusted to us. To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks. There are many such algorithms in the literature, but combining them into a cohesive system is a challenge, as the algorithms must usually be modified to interact properly in a real-world system. In addition, we have found it necessary to invent algorithms of our own. We work hard to avoid unnecessary complexity, but the essential complexity of the task remains high.

Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching a service, we need to reach extremely high confidence that the core of the system is correct. We have found the standard verification techniques in industry are necessary but not sufficient. We routinely use deep design reviews, code reviews, static code analysis, stress testing, and fault-injection testing but still find that subtle bugs can hide in complex concurrent fault-tolerant systems. One reason they do is that human intuition is poor at estimating the true probability of supposedly "extremely rare" combinations of events in systems operating at a scale of millions of requests per second.

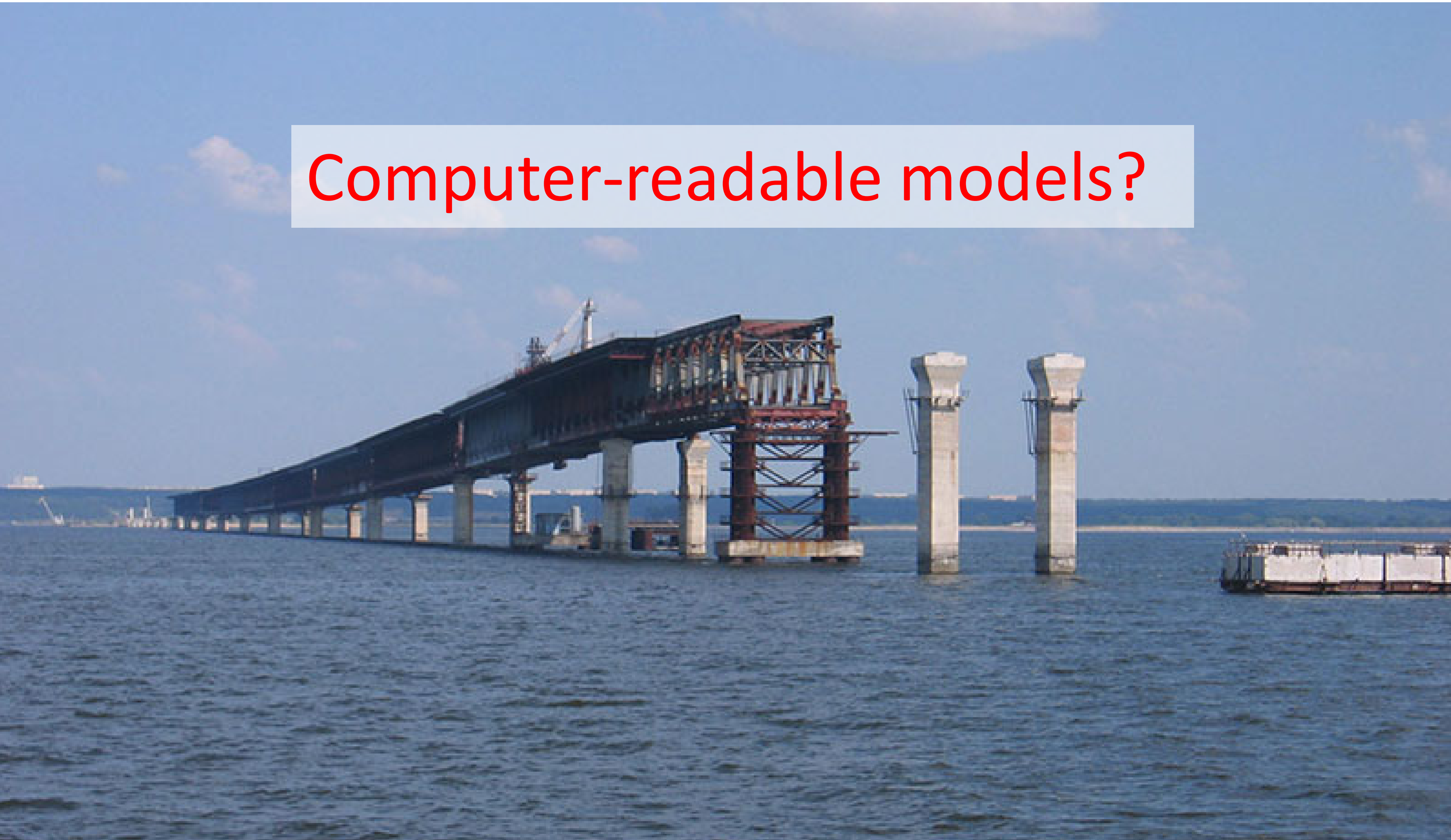
» key insights

- **Formal methods find bugs in system designs that cannot be found through any other technique we know of.**
- **Formal methods are surprisingly feasible for mainstream software development**

Distributed Algorithms and Model Checking

Distributed Algorithms and Model Checking

Distributed Algorithms and Model Checking



Distributed Algorithms and Model Checking

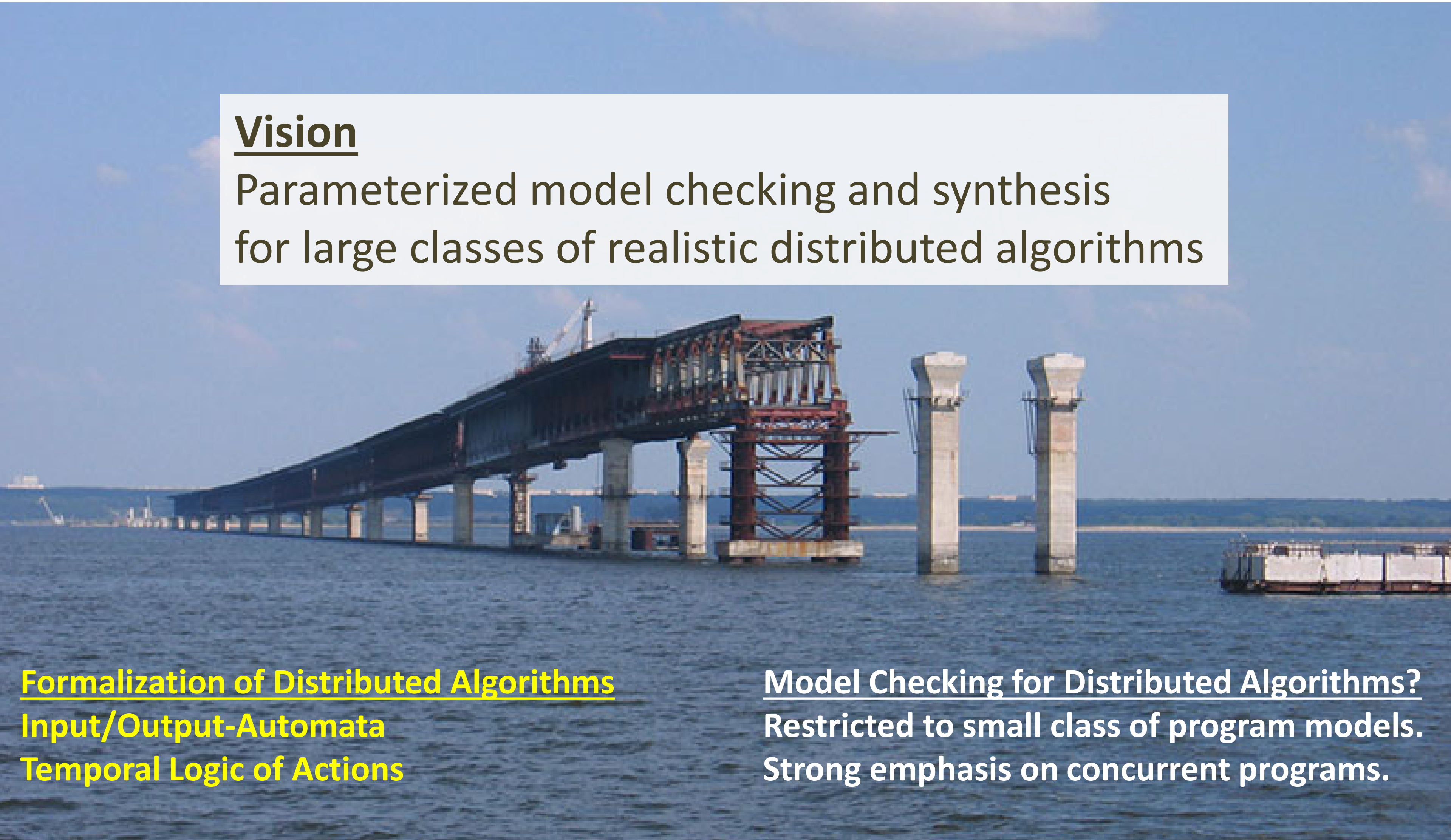
<u>Formalization of Distributed Algorithms</u>

Input/Output-Automata

Temporal Logic of Actions

Distributed Algorithms and Model Checking

Distributed Algorithms and Model Checking



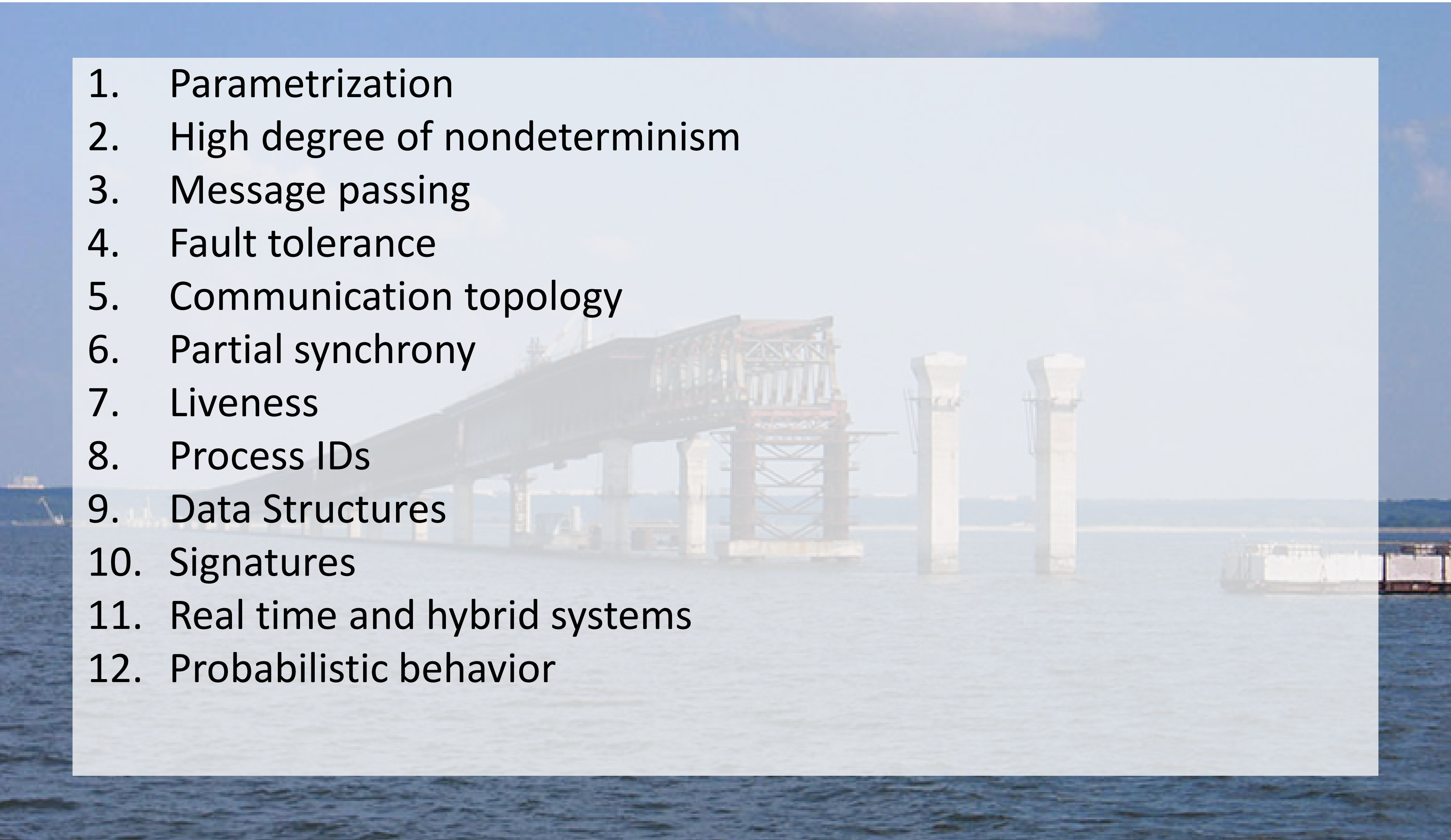
Vision

Parameterized model checking and synthesis
for large classes of realistic distributed algorithms

Formalization of Distributed Algorithms
Input/Output-Automata
Temporal Logic of Actions

Model Checking for Distributed Algorithms?
Restricted to small class of program models.
Strong emphasis on concurrent programs.

Technical Challenges from Distributed Algorithms

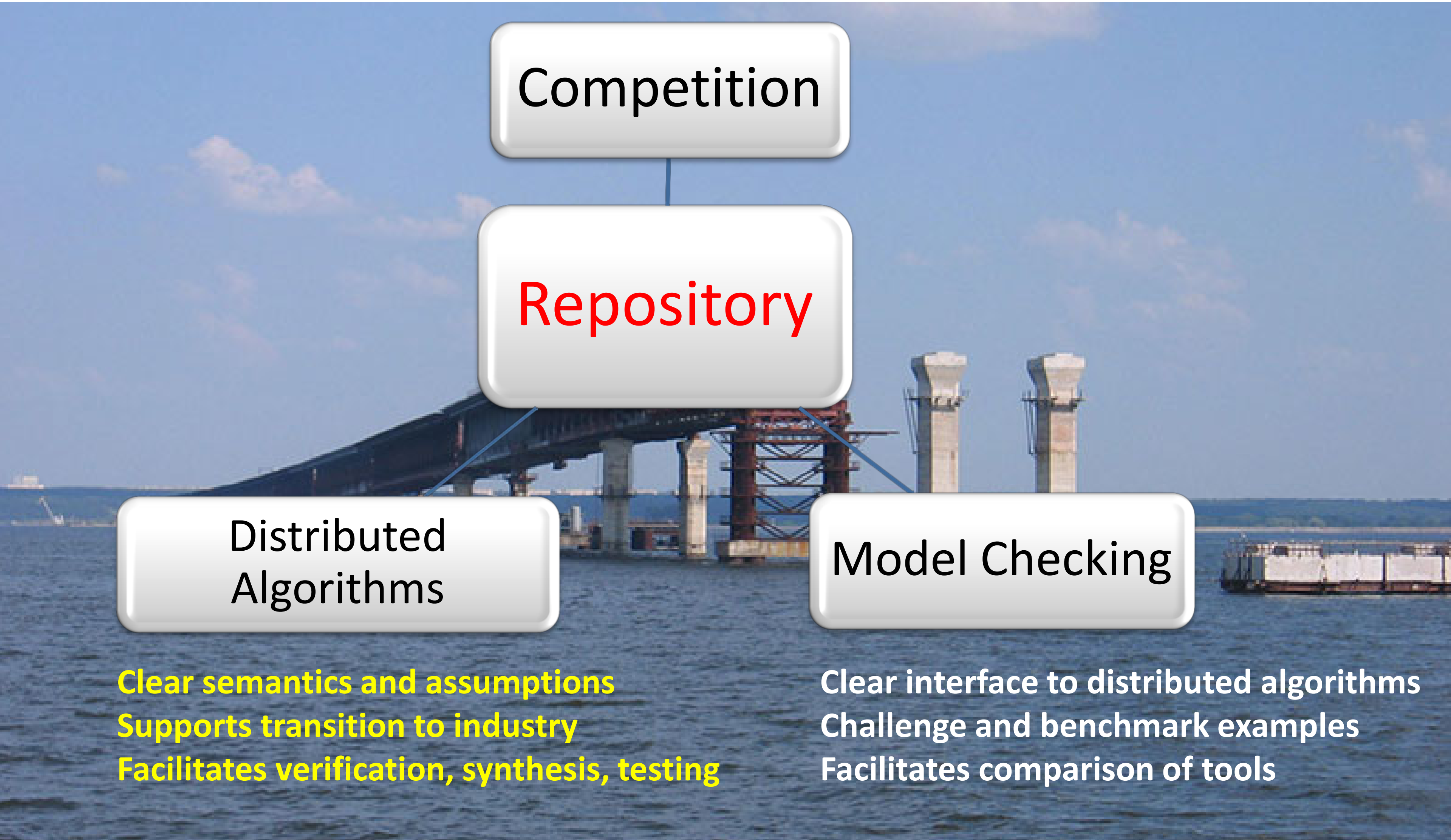


1. Parametrization
2. High degree of nondeterminism
3. Message passing
4. Fault tolerance
5. Communication topology
6. Partial synchrony
7. Liveness
8. Process IDs
9. Data Structures
10. Signatures
11. Real time and hybrid systems
12. Probabilistic behavior

Technical Challenges from Distributed Algorithms

1. Parametrization ☑
2. High degree of nondeterminism ☑
3. Message passing ☑
4. Fault tolerance ☑
5. Communication topology
6. Partial synchrony
7. Liveness ☑
8. Process IDs
9. Data Structures
10. Signatures
11. Real time and hybrid systems
12. Probabilistic behavior

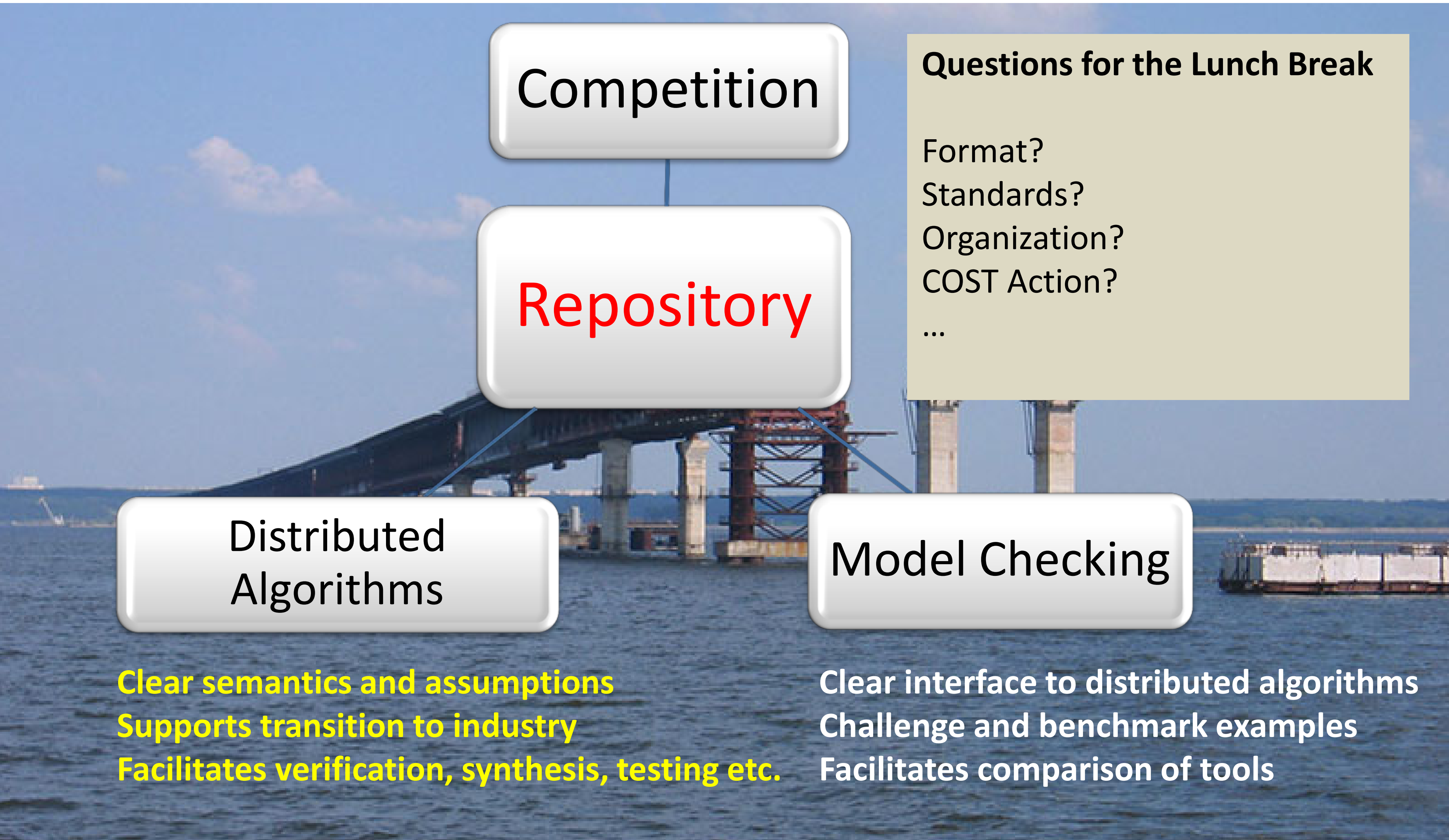
Computer-Readable Models



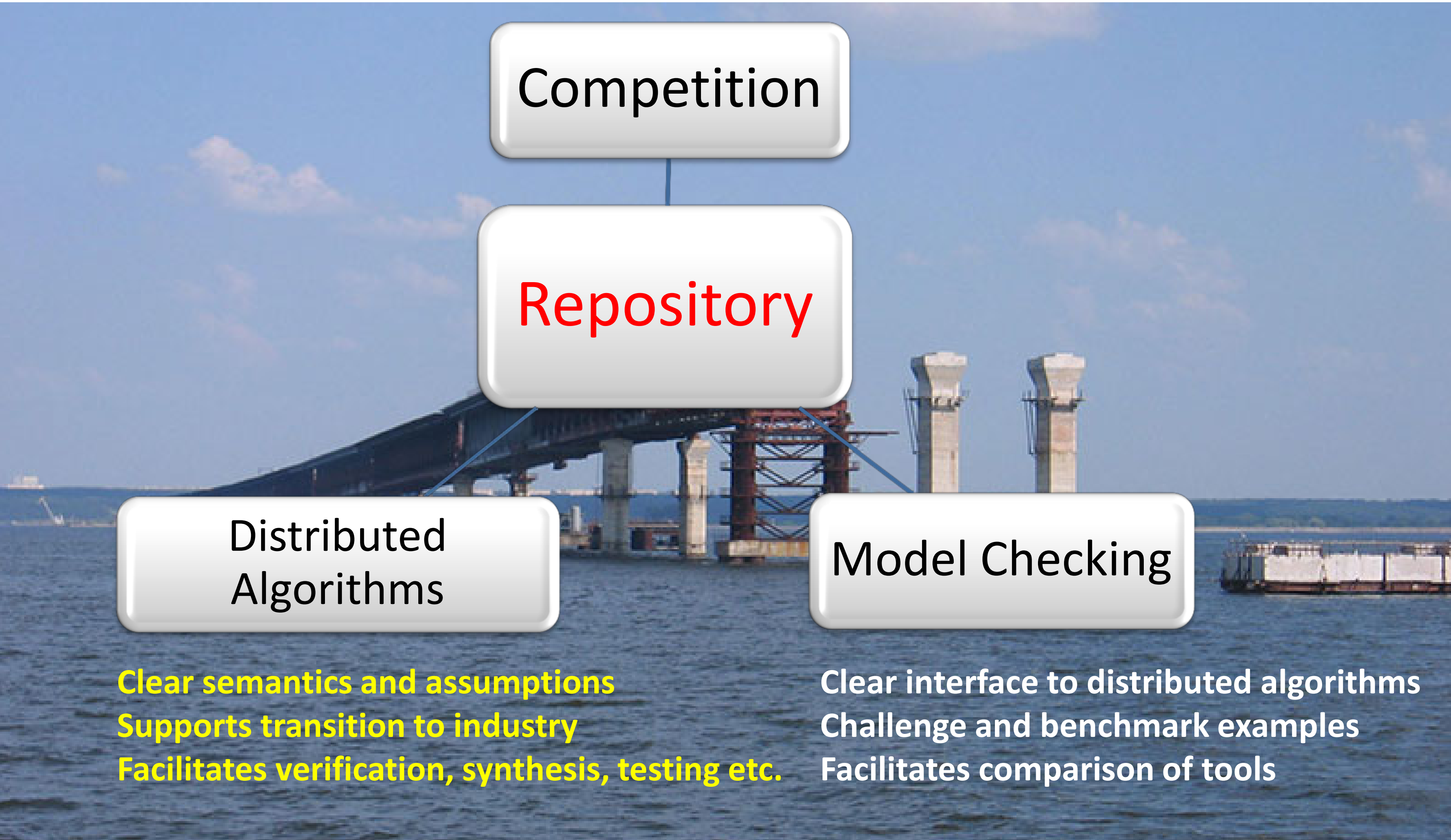
- Clear semantics and assumptions
- Supports transition to industry
- Facilitates verification, synthesis, testing

- Clear interface to distributed algorithms
- Challenge and benchmark examples
- Facilitates comparison of tools

Computer-Readable Models



Computer-Readable Models



Clear semantics and assumptions
Supports transition to industry
Facilitates verification, synthesis, testing etc.

Clear interface to distributed algorithms
Challenge and benchmark examples
Facilitates comparison of tools

Thank you for your attention!