SMT-based approaches to Model Checking of Distributed Broadcast Algorithms

Francesco Alberti, **Silvio Ghilardi**, Andrea Orsini, Elena Pagani
Università degli Studi di Milano / Fondazione Centro San Raffaele

FRIDA 2015

June 5, 2015

Goal of the work

- automatic formal validation of fault-tolerant distributed algorithms
- crucial for many critical applications:
  - industrial plant monitoring via wireless sensors and actuators networks
  - fleet coordination
  - intelligent transport systems
  - aerospace applications
- algorithms involve an arbitrary number N of processes whose behavior depends on the messages they exchange
- algorithms are infinite-state reactive parameterized systems
  - verification that UNSAFE states cannot be reached from initial configurations, for any number of processes
- verification using either array-based systems or counter systems

Goal of the work

The work is part of a wider project of realizing a broad scope SMT-based model-checker able to handle *quantified assertions* over *array* theories.

Goal of the work

The work is part of a wider project of realizing a broad scope SMT-based model-checker able to handle *quantified assertions* over *array* theories.

The model-checker employs backward search with abstraction and acceleration heuristics

Goal of the work

The work is part of a wider project of realizing a broad scope SMT-based model-checker able to handle *quantified assertions* over *array* theories.

The model-checker employs backward search with abstraction and acceleration heuristics

An SMT-solver is used to discharge quantifier-free proof obbligations; quantifiers are preprocessed by the model-checker using both instantiations and quantifier elimination

Goal of the work

The work is part of a wider project of realizing a broad scope SMT-based model-checker able to handle *quantified assertions* over *array* theories.

The model-checker employs backward search with abstraction and acceleration heuristics

An SMT-solver is used to discharge quantifier-free proof obbligations; quantifiers are preprocessed by the model-checker using both instantiations and quantifier elimination

MCMT current release is 2.5.2 (Yices is the underlying SMT-solver); the tool has been succesfully used to verify safety properties of cache-coherence, mutual exclusion, timed, fault-tolerant systems as well as imperative programs over arrays and strings.

Array-based Systems

By a *theory* we mean here a pair $T = (\Sigma, \mathcal{C})$, where Σ is a first-order signature and $\mathcal{C}$ is a class of Σ-structures (called the models of T).

Array-based Systems

By a *theory* we mean here a pair $T = (\Sigma, \mathcal{C})$, where Σ is a first-order signature and $\mathcal{C}$ is a class of Σ-structures (called the models of T).

- Topology of the parameterised system: theory $T_I = (\Sigma_I, \mathcal{C}_I)$
  E.g.: $\mathcal{C}_I$ consists of all (finite) sets, linear orders, forests/trees, graphs, ...

Array-based Systems

By a *theory* we mean here a pair $T = (\Sigma, \mathcal{C})$, where Σ is a first-order signature and $\mathcal{C}$ is a class of Σ-structures (called the models of T).

- Topology of the parameterised system: theory $T_I = (\Sigma_I, \mathcal{C}_I)$
  E.g.: $\mathcal{C}_I$ consists of all (finite) sets, linear orders, forests/trees, graphs, ...
- Data manipulated by the parameterised system: theories $T_E = (\Sigma_{E_i}, \mathcal{C}_{E_i})$
  Usually $\mathcal{C}_{E_i}$ contains just one structure: integers, reals, Booleans, control locations, ...

Array-based Systems

By a *theory* we mean here a pair $T = (\Sigma, \mathcal{C})$, where Σ is a first-order signature and $\mathcal{C}$ is a class of Σ-structures (called the models of T).

- Topology of the parameterised system: theory $T_I = (\Sigma_I, \mathcal{C}_I)$
  E.g.: $\mathcal{C}_I$ consists of all (finite) sets, linear orders, forests/trees, graphs, ...
- Data manipulated by the parameterised system: theories $T_E = (\Sigma_{E_i}, \mathcal{C}_{E_i})$
  Usually $\mathcal{C}_{E_i}$ contains just one structure: integers, reals, Booleans, control locations, ...
- We assume the availability of SMT solvers deciding the satisfiability of quantifier-free formulae modulo T_I and T_{E_i}.

Array-Based Systems

- the sort `INDEX` is constrained by T_I;
- the sort `ELEM` is constrained by T_E;
- the sort `ARRAY` represents arrays of `ELEM` defined on `INDEX`;
- the 'read' operation $\_[\_]$ is added to $\Sigma_I \cup \Sigma_E$;
- the class of models of A_I^E consists of the three-sorted structures whose reducts are models of T_I, T_E and the sort `ARRAY` is interpreted as the set of total functions from indexes to elements and the read operation is interpreted as function application

Array-Based Systems

- An array-based system on A_I^E with array state variable a is the following pair of formulae:

$$\mathcal{S} = \langle I(a), \tau(a, a') \rangle.$$

Array-Based Systems

- An array-based system on A_I^E with array state variable a is the following pair of formulae:

$$\mathcal{S} = \langle I(a), \tau(a, a') \rangle.$$

- A state of an array-based system is an assignment to the variable a in a model of A_I^E

Array-Based Systems

- An array-based system on A_I^E with array state variable a is the following pair of formulae:

$$\mathcal{S} = \langle I(a), \tau(a, a') \rangle.$$

- A state of an array-based system is an assignment to the variable a in a model of A_I^E
- A safety problem for $\mathcal{S}$ is the following: given a formula $K(a)$, is

$$I(a_0) \wedge \tau(a_0.a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge K(a_n)$$

A_I^E-satisfiable for some n?

Revisiting Backward Reachability

Idea: recast symbolically the backward reachability algorithm

function BReach(K)
  $i \longleftarrow 0$; $BR^0(\tau, K) \longleftarrow K$; $K^0 \longleftarrow K$
  if $BR^0(\tau, K) \cap I \neq \emptyset$ **then return** unsafe
  repeat
    $K^{i+1} \longleftarrow \mathrm{Pre}(\tau, K^i)$
    $BR^{i+1}(\tau, K) \longleftarrow BR^i(\tau, K) \cup K^{i+1}$
    if $BR^{i+1}(\tau, K) \cap I \neq \emptyset$ **then return** unsafe
    else $i \longleftarrow i + 1$
  until $BR^{i+1}(\tau, K) \subseteq BR^i(\tau, K)$
  return safe
end

Revisiting Backward Reachability

Idea: recast symbolically the backward reachability algorithm

function BReach(K)
  $i \longleftarrow 0$; $BR^0(\tau, K) \longleftarrow K$; $K^0 \longleftarrow K$
  if A_I^E-check$(BR^0(\tau, K) \wedge I) = \text{sat}$ **then return** unsafe
  repeat
    $K^{i+1} \longleftarrow \text{Pre}(\tau, K^i)$
    $BR^{i+1}(\tau, K) \longleftarrow BR^i(\tau, K) \vee K^{i+1}$
    if A_I^E-check$(BR^{i+1}(\tau, K) \wedge I) = \text{sat}$ **then return** unsafe
    else $i \longleftarrow i + 1$
  until A_I^E-check$(\neg(BR^{i+1}(\tau, K) \rightarrow BR^i(\tau, K)) = \text{unsat}$
  return safe
end

But this is problematic... unless right formats for I, τ, K are found!

Format for initialization formulae

Proposed format for I: $\forall^I$-formulae

$$\forall \underline{i}\ \phi(\underline{i}, a[\underline{i}])$$

where $\underline{i}$ is a tuple of variables of sort `INDEX` and ϕ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula

For instance, the formula $\forall i.\ a[i] = \texttt{idle}$ says that all processes are in state `idle`.

Format for unsafety problems formulae

Proposed format for K: $\exists^I$-formulae

$$\exists \underline{i}\ \phi(\underline{i}, a[\underline{i}])$$

where $\underline{i}$ is a tuple of variables of sort `INDEX` and ϕ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula.

For instance, the formula

$$\exists i_1 \exists i_2.\ (i_1 \neq i_2 \wedge a[i_1] = \texttt{use} \wedge a[i_2] = \texttt{use})$$

expresses that mutual exclusion is violated.

Format for transitions formulae

Proposed format for τ: we use disjunctions of formulae of the kind

$$\exists \underline{i} \left(\phi_L(\underline{i}, a[\underline{i}]) \wedge a' = \lambda j \, F(\underline{i}, a[\underline{i}], j, a[j]) \right) \tag{1}$$

where F is a case-defined function (cases are described by quantifier-free formulae).

Format for transitions formulae

Proposed format for τ: we use disjunctions of formulae of the kind

$$\exists \underline{i} \left(\phi_L(\underline{i}, a[\underline{i}]) \wedge a' = \lambda j \, F(\underline{i}, a[\underline{i}], j, a[j]) \right) \tag{1}$$

where F is a case-defined function (cases are described by quantifier-free formulae).

For instance, the formula

$$\exists i. \left(a[i] = \texttt{use} \wedge a' = \lambda j \, (\mathit{if}\ j = i\ \mathit{then}\ \texttt{idle}\ \mathit{else}\ a[j]) \right)$$

is one of the disjunctions of the transition of the 'bakery' algorithm.

Why it works...

This framework works because:

- closure under preimage of existential formulae guarantees symbolic backward search;
- safety tests are decidable by combination results;
- fixpoints tests are decidable under relatively mild assumptions;
- termination is guaranteed under strong assumptions, but holds in many practical cases.

Why it works...

To improve termination and performances, suitable heuristics are available:

- acceleration (for array fragments too);
- abstraction/refinement loops.

Thes heuristics are both implemented in MCMT since version 2.0, although still not in a full way.

Counters specifications

Array-based systems can implement also counters or shared variables: these can be seen as constant arrays (MCMT specifications accept them directly as global variables of suitable type).

Counters specifications

Array-based systems can implement also counters or shared variables: these can be seen as constant arrays (MCMT specifications accept them directly as global variables of suitable type).

When only counters integer variables occur, array-based systems becomes counter automata (quantifiers are redundant, variable updates and guards become just Presburger definable expressions).

Counters specifications

Array-based systems can implement also counters or shared variables: these can be seen as constant arrays (MCMT specifications accept them directly as global variables of suitable type).

When only counters integer variables occur, array-based systems becomes counter automata (quantifiers are redundant, variable updates and guards become just Presburger definable expressions).

For counters systems, highly engineered SMT-based model checkers, implementing sophisticated abstraction/refinement algorithms like IC3, are available. We used μZ and NuXmv in our experiments.

Counters specifications

Array-based systems can implement also counters or shared variables: these can be seen as constant arrays (MCMT specifications accept them directly as global variables of suitable type).

When only counters integer variables occur, array-based systems becomes counter automata (quantifiers are redundant, variable updates and guards become just Presburger definable expressions).

For counters systems, highly engineered SMT-based model checkers, implementing sophisticated abstraction/refinement algorithms like IC3, are available. We used μZ and NuXmv in our experiments.

It should be noted that whereas array-based specifications are usually quite faithful and close to the original informal specifications, this is not the case for counters specifications: the latter are simulations possibly introducing spurious runs.

1 Specification Formalisms: arrays and counters

2 Case Studies

Reliable Broadcast problem

- group G of N cooperating processes generating messages
- some processes may be subject to failures during the algorithm run
- for each message, the processes must agree on whether to consider it, and deliver it to the user, or not
- more formally, for each message the algorithm must guarantee the following safety property

Agreement

If a correct process decides to deliver a message m, then all correct processes decide to deliver m.

T. D. Chandra and S. Toueg.
Time and message efficient reliable broadcasts.
In *Proceedings of the 4th international workshop on Distributed Algorithms*, 289–303, 1991.

Behavior of faulty processes: preliminaries

- message broadcasting is not atomic with respect to crash failures
- send-omission algorithm as an example:
  - a process may omit to send any message of the algorithm
  - the receiver does not know whether the sender is crashed or not, and what other processes in the group have possibly received the message
- general-omission: a process may also omit to receive some messages
- byzantine: a faulty process may also send messages with a wrong content

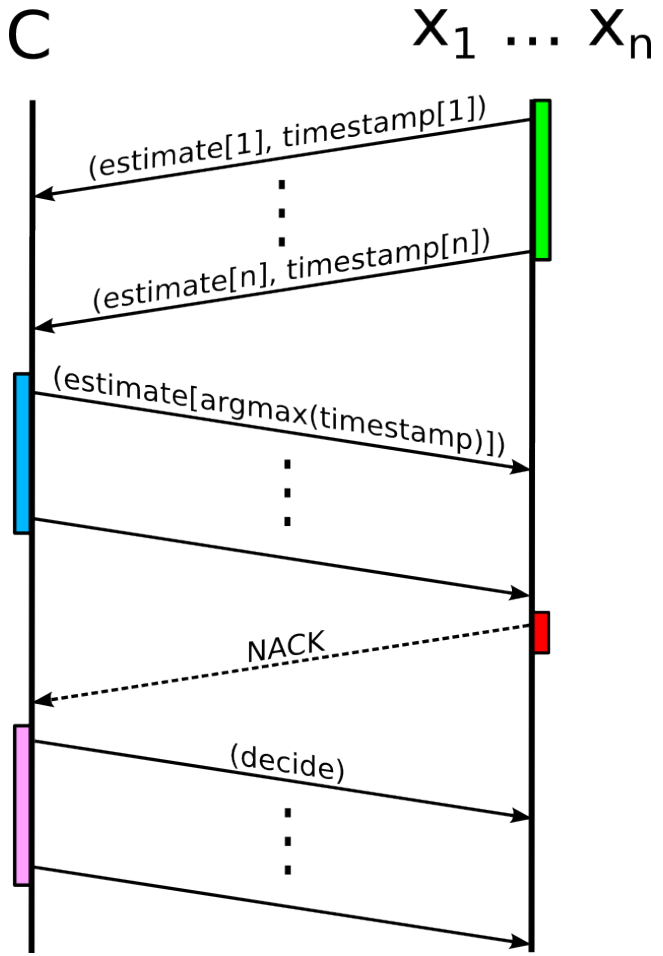
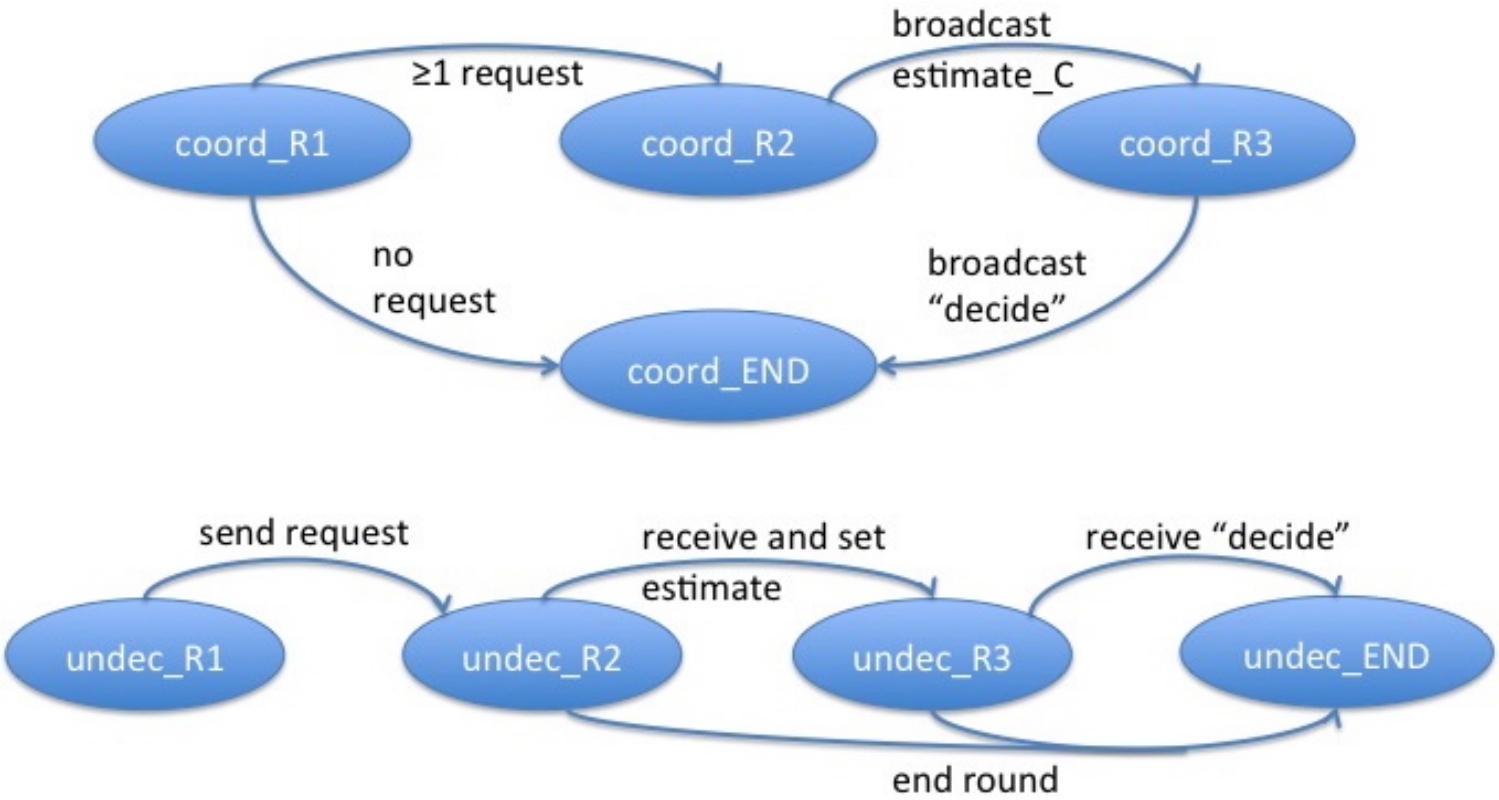


Figure : send-omission algorithm

Scheme of the algorithm for crash failures

- each coordinator tries to stabilize the estimate it owns



Crash failures: pseudo-code

Initialization:
    if (p is the sender)
        then $estimate[p] \leftarrow m$;
        else $estimate[p] \leftarrow \perp$;
    $state[p] \leftarrow undecided$;
End Initialization

for $c \leftarrow 1, 2, ..., N$ **do** // Process c becomes coordinator for three rounds
1. **Round 1:**
2. All *undecided* processes p send *request* to c;
3. **if** (c does not receive any request) **then** it skips rounds 2 and 3;
4. **Round 2:**
5. c multicasts $estimate[c]$;
6. All *undecided* processes p that receive $estimate[c]$ do
7. $estimate[p] \leftarrow estimate[c]$;
8. **Round 3:**
9. c multicasts *Decide*;
10. All *undecided* processes p that receive *Decide* **do**
11. $decision[p] \leftarrow estimate[p]$;
12. $state[p] \leftarrow$ DECIDED;

end for

Crash failures: modelization

Array-based modelization does not offer special difficulties, universal guards can be modeled with an ad hoc quantifier relativization procedure:

F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, G. Rossi
Universal guards, relativization of quantifiers, and failure models in model checking modulo theories,
Journal on Satisfiability, Boolean Modeling and Computation, vol. 8, p. 29-61, 2012.

Crash failures: modelization

Counter modelization is possible: we need an integer variable for the round, an integer variable for the coordinator estimate, integer variables counting decided/undecided processes with estimate $m/\perp$, integer variables counting processes in locations done/undone with estimate $m/\perp$, etc ...

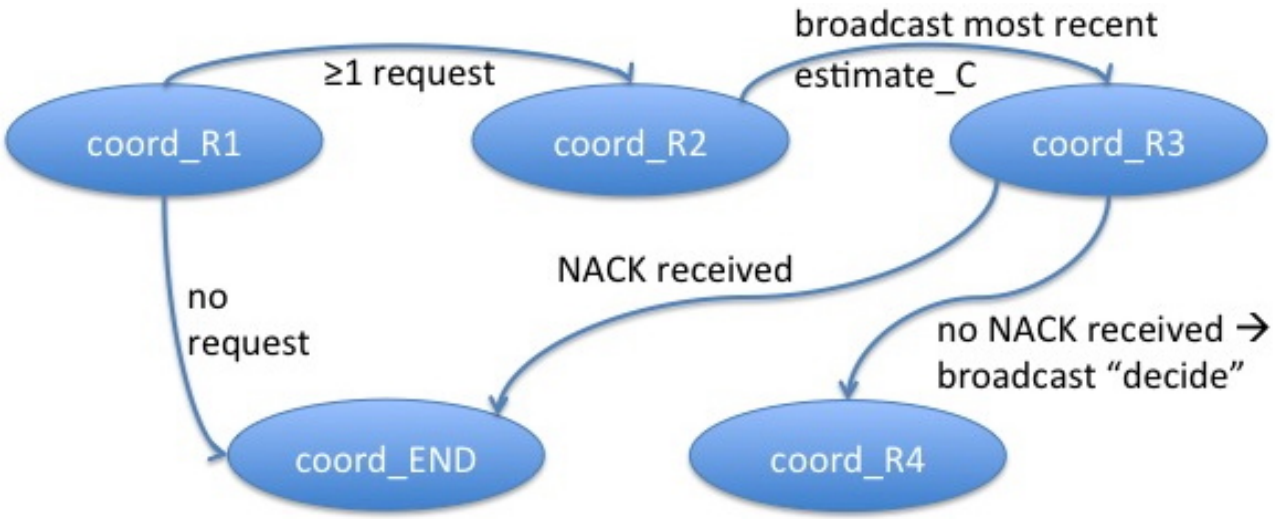
Crash failures: modelization

Counter modelization is possible: we need an integer variable for the round, an integer variable for the coordinator estimate, integer variables counting decided/undecided processes with estimate $m/\perp$, integer variables counting processes in locations done/undone with estimate $m/\perp$, etc ...

We loose the information that a process has already been coordinator, so the simulation with counters allows extra runs where a process has been coordinator more than once.

Scheme of the algorithm for send-omission failures

- undecided processes send the most recent estimate they received, with the indication of the sending coordinator
- they send a `nack` if they do not receive the estimate of the current coordinator, to notify it that it is faulty



Crash and Send-Omission failures: pseudo-code

Initialization:
  if (p is the sender)
    then $estimate[p] \leftarrow m$; $coord\_id[p] \leftarrow 0$;
    else $estimate[p] \leftarrow \perp$; $coord\_id[p] \leftarrow -1$;
  $state[p] \leftarrow undecided$;
End Initialization
for $c \leftarrow 1, 2, \ldots, N$ **do** // Process c becomes coordinator for four rounds
1. **Round 1:**
2. All *undecided* processes p send *request* ($estimate[p]$, $coord\_id[p]$) to c;
3. **if** (c does not receive any request) **then** it skips rounds 2 to 4;
4. **else** $estimate[c] \leftarrow estimate[p]$ with largest $coord\_id[p]$;
5. **Round 2:**
6. c multicasts $estimate[c]$;
7. All *undecided* processes p that receive $estimate[c]$ do
8. $estimate[p] \leftarrow estimate[c]$ **and** $coord\_id[p] \leftarrow c$;
9. **Round 3:**
10. All *undecided* processes p that do not receive $estimate[c]$ send(NACK) to c;
11. **Round 4:**
12. **if** (c does not receive any NACK) **then** c multicasts *Decide*; **else** c HALTS;
13. All *undecided* processes p that receive *Decide* **do**
14. $decision[p] \leftarrow estimate[p]$;
15. $state[p] \leftarrow$ DECIDED;
end for

Send-Omission failures: modelization

Again, array-based specification does not offer problems.

Send-Omission failures: modelization

Again, array-based specification does not offer problems.

Counter specification requires some manipulation, because there is no way of formalizing the meaning of "larger coord-id[p]"

Send-Omission failures: modelization

Again, array-based specification does not offer problems.

Counter specification requires some manipulation, because there is no way of formalizing the meaning of "larger coord-id[p]"
We adjust the situation by using part of our knowledge of the algorihm: the new coordinate estimate can be arbitrary in *some* circumstances ....

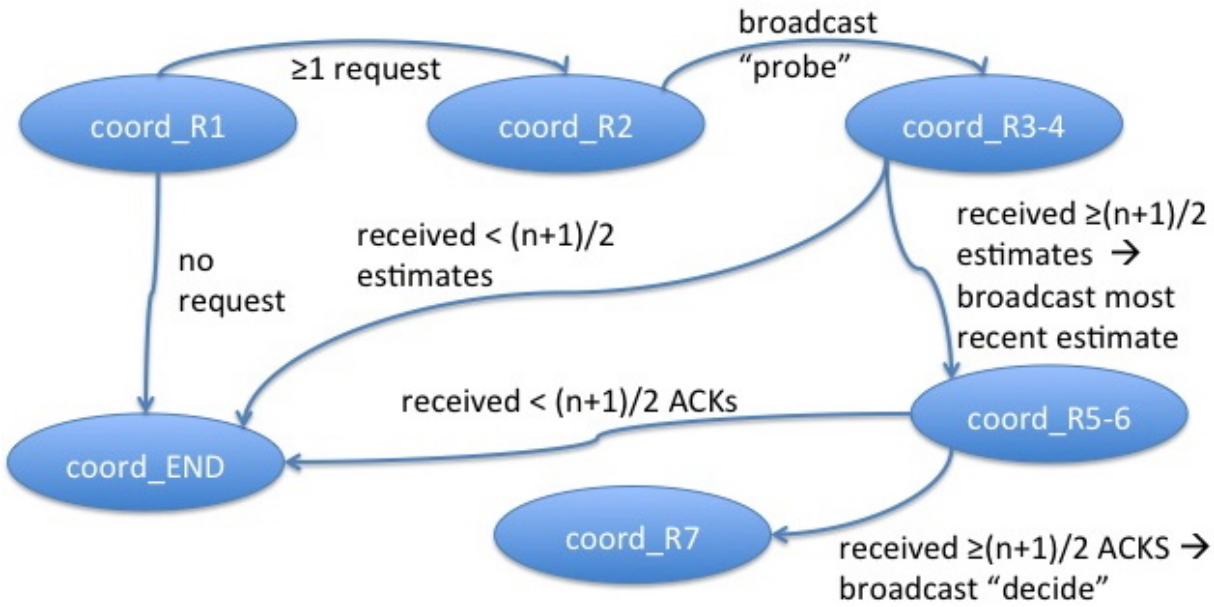
Send-Omission failures: modelization

Again, array-based specification does not offer problems.

Counter specification requires some manipulation, because there is no way of formalizing the meaning of "larger coord-id[p]"
We adjust the situation by using part of our knowledge of the algorihm: the new coordinate estimate can be arbitrary in *some* circumstances ....
This modified specification includes all concrete runs (together with spurious ones).

Scheme of the algorithm for general-omission failures

- if a coordinator receives requests from undecided processes, it asks both decided and undecided processes to send the most recent estimate they received, with the indication of the sending coordinator
- majority resilience: at least $(n+1)/2$ processes must behave correctly throughout the whole algorithm run
  - if the coordinator receives only a minority of feedbacks, it detects its own failure



General Omission failures: modelization

The new features arising here are resilience guards. These are easy to model in counter specifications.

General Omission failures: modelization

The new features arising here are resilience guards. These are easy to model in counter specifications.

In principle, array-based formalisms support this: one uses suitable transitions loops like

$$\texttt{int I, J} = 0;$$
$$\texttt{for(I} = 0; \texttt{I} \leq \texttt{N}; \texttt{I} + +)\{\texttt{if(received\_from[I]} == 1)\texttt{J} + +; \}$$

and then uses the value of J in resilience guards.

General Omission failures: modelization

The new features arising here are resilience guards. These are easy to model in counter specifications.

In principle, array-based formalisms support this: one uses suitable transitions loops like

$$\texttt{int I, J} = 0;$$
$$\texttt{for(I} = 0; \texttt{I} \leq \texttt{N}; \texttt{I} + +)\{\texttt{if(received\_from[I]} == 1)\texttt{J} + +; \}$$

and then uses the value of J in resilience guards.
In practice, the actual heuristics for preventing termination implemented in MCMT may fail to succeed when such solutions are adopted. We believe there is room for improvement, though.

Byzantine broadcast primitive

- substitutes authentication obtained via unforgeable signatures
- resilience: at most (n -1)/3 of the processes may fail

T.K. Srikanth and S. Toueg.
Simulating authenticated broadcasts to derive simple fault-tolerant algorithms.
Distributed Computing, 2(2):80–94, 1987.

Byzantine broadcast primitive: pseudo-code

Round 1:

Phase 1: process p sends Init(p,m,1);

Phase 2: $\forall$ process does:

if (received Init(p, m, 1) from p in Phase 1)

then send Echo(p, m, 1) to all;

if (received Echo(p, m, 1) from $\geq n - t$ distinct processes in Phase 2)

then Accept(p, m, 1);

Round 2:

$\forall$ Phase, $\forall$ process does:

if (received Echo(p, m, 1) from $\geq n - 2t$ distinct processes in previous phases $\wedge$ not sent echo yet)

then send Echo(p, m, 1) to all;

if (received Echo(p, m, 1) from $\geq n - t$ distinct processes in this and previous phases)

then Accept(p, m, 1);

- different algorithms structure: processes consider the cumulative number of Echo's messages received so far

Byzantine broadcast primitive

These are the properties we would like to verify:

Correctness

If correct process p broadcasts (p,m,k) in round k, then every correct process accepts (p,m,k) in the same round.

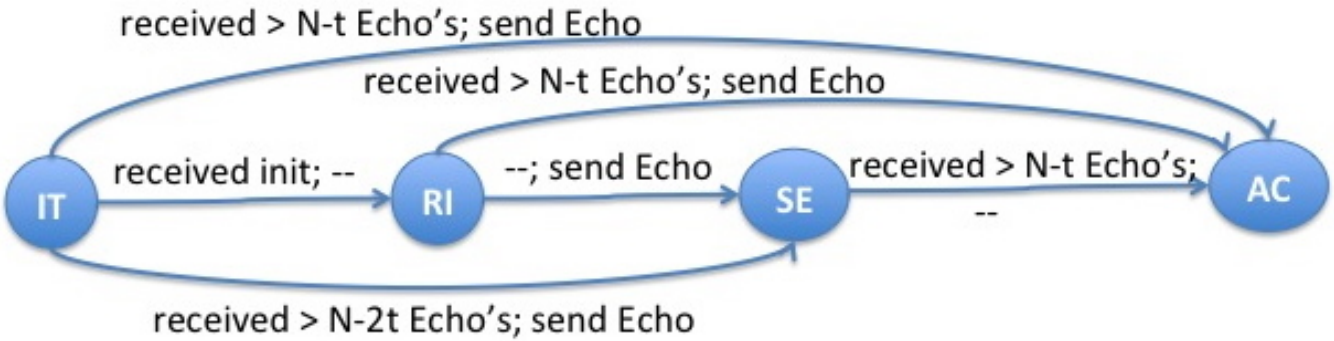
Unforgeability

If process p is correct and does not broadcast (p,m,k), then no correct process ever accepts (p,m,k).

Relay

If a correct process accepts (p,m,k) in round $r \geq k$ then every other correct process accepts (p,m,k) in round $r + 1$ or earlier.

Outline of the modelization



- We model faulty processes through their effect on correct ones.
- We omit reference to the round when processes accept.
- Correctness and Relay must be reformulated as safety properties (in particular Relay is broken into two parts).

Outline of the modelization

For array-based specifications, we used the above suggested methods for formalizing resilience conditions. MCMT was able to solve the problems with suitable settings of abstraction heuristics.

Outline of the modelization

For array-based specifications, we used the above suggested methods for formalizing resilience conditions. MCMT was able to solve the problems with suitable settings of abstraction heuristics.

For counter specifications, some specific simulation abstraction was needed (eg to simulate non symmetric faults, 'message withdrawns' were allowed in interleaving with each process actions). All our tools solved the problems easily.

Performance evaluation

For counters specifications, we have implemented translators from MCMT to Z3 and nuXmv in order to guarantee fair comparison (on the other hand, array and counters specifications are not directly comparable).

	MCMT			Z3	nuXmv
File	Depth	Nodes	Time	Time	Time
crash.mcmt	8	39	0.19"	0.69"	0.36"
sndom.mcmt	21	1772	77"	846"	22"
genom.mcmt	42	10102	6175"	t.o.	t.o.
byz_unforg.mcmt	7	51	0.42"	0.19"	0.04"
byz_corr.mcmt	7	131	6.16"	0.13"	0.24"
byz_relayA.mcmt	6	38	0.23"	0.07"	0.13"
byz_relayB.mcmt	8	185	5.21"	0.05"	0.16"
crash_array.mcmt	13	113	0.44"		
sndom_array.mcmt	39	11206	2317"		
unforg_array.mcmt	15	446	3.57"		
corr_array.mcmt	3	22	0.04"		
relayI_array.mcmt	2	38	0.05"		
relayII_array.mcmt	4	21	0.02"		

Concluding remarks

- The feeling is that array-based specifications of challenging problems are more difficult to verify.

Concluding remarks

- The feeling is that array-based specifications of challenging problems are more difficult to verify.
- Nevertheless, such specifications are more faithful, fine-grained, and expressive.

Concluding remarks

- The feeling is that array-based specifications of challenging problems are more difficult to verify.
- Nevertheless, such specifications are more faithful, fine-grained, and expressive.
- Counter specifications can be discharged by efficient solvers, so such specifications might produce useful invariants inside array-based or user-assisted verification.