SMT-based approaches to Model Checking of Distributed Broadcast Algorithms: some case studies

Francesco Alberti
alberti.francesco@hsr.it, http://www.falberti.it
Fondazione Centro San Raffaele
Milano, Italy
Silvio Ghilardi
silvio.ghilardi@unimi.it, http://users.mat.unimi.it/users/ghilardi/
Università degli Studi di Milano
Milano, Italy
Andrea Orsini
andrea.orsini2011@gmail.com
Università degli Studi di Milano
Milano, Italy
Elena Pagani
pagani@di.unimi.it, http://www.di.unimi.it/pagani
Università degli Studi di Milano
Milano, Italy
June 19, 2015

Abstract

The automated, formal verification of distributed algorithms is a crucial, although challenging, task. The processes executing these algorithms communicate to one another, their actions depend on the messages received, and their number is arbitrary. These characteristics are captured by so-called reactive parameterized systems. The task of validating or refuting properties of these systems is daunting, due to the difficulty of limiting the possible evolutions, thus having to deal with genuinely infinite-state systems.

In this paper, we study the properties of distributed algorithms solving the reliable broadcast problem in various failure models. We investigate the suitability of a direct Satisfiability Modulo Theories (SMT) approach to model these algorithms in order to validate safety properties. In a previous work, we modeled distributed algorithms using the declarative framework of array-based systems. In this work, we try also a simulation of array-based systems via counter systems. Our experiments show that this simulation does not indeed introduce spurious runs violating the safety properties we want to formally verify in a class of problems which is even larger than that considered in our previous work.

We report the related performance evaluations of some SMT-based model-checkers (essentially, our tool MCMT and tools like $\mu$Z, nuXmv). The experimental results are interesting because they show on one hand that state-of-the-art SMT-based technology can handle problems arising in fault-tolerant environments, and on the other hand that different heuristics and search strategies (e.g. acceleration versus abstraction) can have practical impact.

Keywords: reactive parameterized systems; SMT-based model checking; fault-tolerant broadcast algorithms; array-based systems; counter systems.

*Reference author
1 Introduction

The validation of fault-tolerant distributed algorithms like those in [22, 5] is a crucial, although challenging, task. This kind of algorithms may support coordinated actions of distributed systems in critical applications such as, e.g., industrial plant monitoring through wireless sensors and actuators networks, fleet coordination, intelligent transport applications, or aerospace applications. Hence, guaranteeing certain safety properties of the algorithms is compulsory. The processes executing these algorithms communicate to one another, their actions depend on the messages received, and their number is arbitrary. These characterisitics are captured by so called reactive parameterized systems, that is, systems composed by an arbitrary number $N$ of processes (“parameterized”), whose behavior depends on the interactions with the environment (“reactive”), namely, with the other processes.

We are interested in formally validating or refuting safety properties of reactive parameterized systems encoding fault-tolerant algorithms. This is a daunting task given their intrinsic infinite-state nature. In this paper, we consider in particular distributed algorithms to solve the reliable broadcast problem (see, e.g., [21]) in the presence of crash, send-omission, general-omission, and byzantine failures. In particular, our goal is to analyze the extent to which a direct encoding into Satisfiability Modulo Theories (SMT) problems can be adopted for automated formal verification of these algorithms.

We make our formalizations according to two paradigms: array-based systems and counter systems. The former, implemented in our tool MCMT - ‘Model Checker Modulo Theories’ [15], uses unbounded arrays to represent the states of the processes. The latter uses integer variables essentially to count the number of processes satisfying given properties.

The algorithms are modeled according to both paradigms. However, whereas array-based modelizations are sufficiently faithful and close to the original informal specifications drawn from the literature, the same does not happen for counter systems: the latter can only simulate the original algorithms and such simulation may sometimes be the fruit of an a-priori reasoning on the characteristics of the algorithm, embedded into the model. Despite this fact, all runs from the array-based specifications are represented in the simulations with counters systems (this is in fact the formal content of the notion of a ‘simulation’), thus safety certifications for the simulating model apply also to the original model. The advantage of this approach is that, as it is evident from our experiments, verification of counters systems performs better and is much more supported from the existing technology (as far as we know, MCMT is almost the only tool supporting array-based specifications). In conclusion, whereas array-based specifications remain one of the mainstreams for verification of distributed algorithms (due to their expressivity), a flexible approach should not disregard counter specifications as powerful tools to check subgoals and to solve subproblems fitting a more restrict syntax.

The paper is organized as follows: in Section 2, we report some generalities about model-checking of infinite state systems and about our approaches. In Section 3, we introduce reliable broadcast problems and comment on crucial aspects of their modelizations with counter systems, compared with array-based systems presented in a previous work [1]. Finally, in Section 4 we report a performance evaluation of the computation with the different models: for array-based systems only MCMT is available, whereas for counters systems we compare it with state-of-the-art tools like $\mu$Z and nuXmv. When considering experimental results, we also briefly comment on the impact of the two different heuristics used to prevent divergence (namely abstraction and acceleration).

2 Preliminaries

The behavior of a reactive system can be modeled through a transition system, which is a triple $T = (W, W_0, R)$ such that $W$ is the set of possible configurations of the system – expressed in terms of the state of each component process – $W_0 \subseteq W$ is the set of initial configurations, $R \subseteq W \times W$ is the transition relation: $w_1 R w_2$ (with $w_1, w_2 \in W$) describes how the system may evolve in one step.

A safety problem for a subset $Bad \subseteq W$ consists in determining whether there is a path

$$w_0 R w_1 R w_2 \cdots R w_n$$

within $(W, R)$ leading from $w_0 \in W_0$ to $w_n \in Bad$. We say that $(W', W'_0, R')$ (with safety problem $Bad'$) simulates $(W, W_0, R)$ (with safety problem $Bad$) iff there is a relation $\rho \subseteq W \times W'$ such that (i) for all $w \in W$
there is \( w' \in W' \) such that \( w'w' \); (ii) if \( w'w' \) and \( w \in W_0 \) then \( w \in W'_0 \); (iii) if \( w'w' \) and \( w \in \text{Bad} \) then \( w' \in \text{Bad}' \); (iv) if \( w'w' \) and \( wRv \) there is a path \( w'R'w' \cdots w'R'v' \) with \( v'v' \). It is evident that the existence of such a simulation \( \rho \) guarantees that if there is no path in \((W',W'_0,R')\) leading from \( W'_0 \) to \( \text{Bad}' \), then there is no path in \((W,W_0,R)\) leading from \( W_0 \) to \( \text{Bad} \). Thus one can model-check \((W',W'_0,R')\) (w.r.t. \( \text{Bad}' \)) instead of \((W,W_0,R)\) (w.r.t. \( \text{Bad} \)), in case there is a simulation of the former to the latter (notice however that the existence of a path in \((W',W'_0,R')\) from \( W'_0 \) to \( S' \) does not imply that an analogous path exists in \((W,W_0,R)\) because the latter essentially has ‘more runs’, i.e. ‘more paths’). When we speak of simulations in the paper, we refer to the above notion of simulation (in fact, checking the existence of such a relation satisfying conditions (i)-(iv) above in our examples is a matter of rather straightforward details).

Various approaches are studied in the literature in order to formally verify safety properties. The problems we address in this paper are typically infinite-state: although the behavior of a single process could often be described by a finite state automaton, the family of systems is infinite due to the parameter \( N \in \mathbb{N} \) indicating the number of the component processes. In the infinite-state case an exhaustive search through the states (in the style of textbooks like [8]) is not possible; states must be handled symbolically through logical formulæ. In addition, satisfiability tests for these formulæ need to be discharged when a model-checker performs its analysis; these satisfiability tests are typically constrained by theories describing both data (integers, Booleans, reals, ...) or datatypes (arrays, lists, ...). This is where SMT solvers may be of help.

Essentially, a SMT-solver (Z3 [9], Yices [12], MathSAT [6], CVC4 [3],...)) is an integrated framework for automated reasoning, involving a SAT-solver, a congruence closure solver, a solver for the manipulation of arithmetic expressions, and so on. Among further theories that might be supported by such solvers we have arrays, bit-vectors, non-linear arithmetic, etc.; quantified formulæ are occasionally supported too, but with limitations due to well-known undecidability results.

By itself, an SMT-solver cannot solve a model-checking problem, but a main application of SMT-solvers is within model-checking frameworks, where the model-checker (acting as a client) asks the SMT-solver (acting as a server) to discharge the satisfiability tests it generates. First of all, the transition system must be specified via a logical formalism and the choice of the appropriate formalism requires a careful balance between expressivity and efficiency. When a logical formalism is chosen, the transition system is specified via a set of variables \( \mathcal{X} \), the initial and unsafe configurations are specified via formulæ \( W_0(\mathcal{X}) \), \( \text{Bad}(\mathcal{X}) \) and the transition relation is specified via a formula \( R(\mathcal{X}, \mathcal{X}') \). When this formal framework is fixed, the SMT-solver can be used for instance as follows. If we are supplied an invariant \( I(\mathcal{X}) \), the SMT-solver can check whether \( \text{Bad}(\mathcal{X}) \land I(\mathcal{X}) \) is satisfiable, thus proving in the negative case that states in \( \text{Bad} \) are not reachable. The SMT-solver can also certify that \( I \) is actually an invariant, by checking formally that the initial set of states is included in \( I \) and that the system cannot exit \( I \) (starting from a set in \( I \)) when executing a step of the transition \( R \). All this amounts to check the unsatisfiability of the following two formulæ

\[
W_0(\mathcal{X}) \land \neg I(\mathcal{X}) \quad I(\mathcal{X}) \land R(\mathcal{X}, \mathcal{X}') \land \neg I(\mathcal{X}')
\]

Even when the above satisfiability tests are effective and feasible (because the underlying logic is not too expressive), the trouble is that invariants can be far from trivial and one cannot expect a user to be able to supply them in full details.

To sum up, the problems we need to face when trying to use an SMT-based approach to model-checking are two-fold: (i) choosing a formalization; (ii) defining a search strategy for invariants. Whereas (ii) typically depends on the technology implemented in each model checker (we shall briefly turn on this below too, when we describe the tools), (i) is a modelization problem relying on the user’s choices: we discuss our own choices in next subsection.

### 2.1 Choosing a formalization

In this paper, distributed algorithms will be described, for symbolic model checking, using two methods. First, we have the array specification relying on the notion of an array-based system [14]. An array-based system is a transition system where a configuration, or system state, is represented as a collection of arrays whose length is not a priori bounded and whose \( i \)-th elements represent the state of process \( p_i \). Array-based systems can be enriched by adding to variables representing arrays also variables representing shared data, like integers, Booleans, etc. This is the more natural and widely applicable approach, and is well suited for
parameterized systems; this formalization is adopted by tools like MCMT [15] and Cubicle [7]. Relevant properties need quantifiers to be expressed: for instance, the violation of mutual exclusion is expressed as

$$\exists z_1 \exists z_2 \ (z_1 \neq z_2 \land a[z_1] = \text{critical} \land a[z_2] = \text{critical})$$

(1)

where $a$ maps every process to its location.

Counter systems is the other formalization we take into consideration in this work. In a counter system, we can only use integer variables to describe the system state, i.e., we can just count the number of processes which are in each possible location (or, more generally, satisfying a predicate). The method is suitable in case the behavior of the processes is driven by conditions of the type:

```
if (number of received messages is in [minThreshold, maxThreshold])

    then perform action;
```

where ‘action’ can be a location change, a broadcast send, etc. The counters specification may be used when processes are indistinguishable, and it does not matter “who” performs an action but rather “how many” do it (so called threshold-guarded algorithms [17, 18]). Counter specifications have been used to formalize broadcast protocols [13, 10] and are the basic formalism accepted by automata-based tools like FAST [2]. The main advantage of counter specifications is that only integer variables are used; quantifiers are not needed (for instance in this setting 1 can be formulated simply as critical > 1), so much more support is available from existing solvers. The drawback is that this approach may not be expressive enough - when processes are structured as a ring, as a linear order or as a graph, nothing can be done within it. Even in case processes are unstructured, the use of counters models may require some form of abstraction. Take for instance a byzantine environment: correct processes either send or do not send a message to all other processes, but byzantine faulty processes may behave non-symmetrically, i.e. they may maliciously send messages only to a subset of processes. In this case, it is not sufficient just to count the number of processes having sent a message. This problem may be overcome by introducing some form of abstraction (see below, or see the interval abstractions of [17] for another solution); abstractions, however, may cause spurious behavior, so in principle there is no guarantee that counters formalizations may work when they are combined with such abstractions. Otherwise said, abstractions produce just simulations, in the formal sense explained above.

On the other hand, in order for array-based model-checkers to be able to perform in a natural way arithmetic reasoning about the cardinality\(^1\) of the set of processes satisfying certain properties, an integrated framework (combining cardinality constraints and array-like specifications) would be desirable. The problem is not that of changing the formal model (array-based systems are appropriate), but to enrich the logic supported to reason about such formal model. The enrichment should be appropriately designed to combine expressivity with computational efficiency (see [11] for an attempt in this direction).

### 2.2 The tools

For array specifications, we use the Model Checker Modulo Theories (MCMT) [15]; for counter specifications, it is possible to use also other SMT-based model-checkers (we choose $\mu$Z [16] and nuXmv [4]) because counter specifications only involve integer variables.

- MCMT is a SMT-based model checker able to verify the properties of infinite-state reactive parameterized systems. MCMT exploits Yices [12] as the underlying SMT-solver. Yices is an SMT solver written in C and maintained by SRI International; it supports all the theories in SMT-LIB, and is able to manipulate also lambda expressions, recursive datatypes, and records. MCMT uses backward reachability search. It has suitable options for abstraction/refinement search; however, MCMT abstraction is tailored to array-based systems and so MCMT abstraction is too peculiar to be really operative in case of pure counter specifications. MCMT implements also some form of acceleration (both for arrays and for integer variables); using acceleration the tool may be able to describe in one shot the effect of executing a loop any finite number of times.

- $\mu$Z is the Z3 module operating with Horn clauses specifications; the formalism of Horn clauses is becoming a common specification language for model checkers. Using this formalism, we can specify the safety

\(^1\)This is required, for example, to express resilience properties, i.e., assumptions of the kind “at most $k$ processes will fail”, where $k$ is a symbolic constant (see below for a more detailed discussion on this).
problems addressed in the counter specifications of this paper. As a search mechanism, $\mu Z$ employs an IC3-based algorithm - a sophisticated abstraction/refinement search driven by the property to be checked.

- muXmv is the evolution for infinite-state systems of the symbolic model checker NuSMV; it uses MathSat as the underlying SMT-solver and implements state-of-the-art abstraction/refinement algorithms like IC3.

Of course, a comparison of the different tools is not easy and is not our main goal. Our aim is more modest: we just want to show that, by introducing rather natural and simple modeling techniques, current SMT-technology can successfully deal with our benchmarks.

3 Considered Class of Problems

In this section, we discuss the modeling of fault-tolerant distributed algorithms using counter systems. We consider four algorithms, namely, the reliable broadcast algorithms for crash (CBA), send-omission (SOBA), or general-omission (GOBA) failures described in [5], and the algorithm for a broadcast primitive with byzantine failures (BBP) described in [22]. We discuss just the relevant aspects of modelization; interested readers may find the complete models at http://users.mat.unimi.it/users/orsini/dbaMC_experiments.html. In the next section, we report some performance evaluation compared against modeling via array-based systems.

3.1 CBA, SOBA, and GOBA algorithms

Let $G$ be a group of cooperating processes. The Reliable Broadcast problem requires that, when a process in $G$ sends a message $m$, either all the correct processes in $G$ deliver $m$ to their users, or none of them delivers $m$, in spite of failures. In CBA, processes may prematurely stop performing any action; the halt may occur in the middle of a broadcast. In SOBA, processes may either crash, or transiently omit to send some messages requested by the algorithm. In GOBA, processes in addition may transiently omit to receive some messages.

A correct process is a process that does not fail in any way for the whole algorithm run. More formally, an algorithm correctly solves the Reliable Broadcast problem if it satisfies the Agreement property:

**Agreement:** If a correct process decides to deliver a message $m$, then all correct processes decide to deliver $m$.

To guarantee the algorithm correctness, up to $t$ processes may fail (resilience), with $t \leq N - 1$ for CBA and SOBA, and $t \leq (N - 1)/2$ for GOBA. The system must be synchronous, in the sense that both the time for a message to arrive to its destination, and the time for a process to execute an algorithm step, are upper bounded. Hence, the algorithms evolve in rounds whose length is finite. For the sake of space, in this work we focus on the modelization of SOBA through counter systems; similar mechanisms are used for GOBA, while CBA is a special case. The pseudo-code for both CBA and SOBA is reported by Algorithm 1: by ignoring the underlined instructions, CBA is obtained; for SOBA, consider all the instructions in the pseudocode regardless of the fact that they are underlined or not. In [1], the discussion of how to model both CBA and SOBA using the array-based method is reported.

The counters modelization uses only global variables that count how many processes are in a certain state. In particular, we count (i) the number of either correct or faulty undecided processes owning or not the message $m$ to be reliably broadcast to the group of processes ($\text{undC}/F(m/\perp)$), (ii) the number of either correct or faulty processes that have decided ($\text{decC}/F(m/\perp)$), (iii) the round number, and (iv) the number of either requests or nacks received by the current coordinator. Initially, the round number is 1, no process has undertaken any action, and only the sender – which is also the first coordinator – owns $m$. This is described by initializing ($\text{undC}(m) + \text{undF}(m)) = 1$. The unsafe formula to be refuted, negating the Agreement property, is expressed as: $\varphi = (\text{decC}(m) > 0 \land \text{decC}(\perp) > 0)$.

In Round 1, every type and number of undecided processes may send a request, and the steps are freely interleaved. As an example, the guard habituating the transition of undecided correct processes with no message is:
Algorithm 1 Pseudo-code for CBA and SOBA

Initialization:

if (p is the sender)
  then estimate[p] ← m; coord_id[p] ← 0;
else estimate[p] ← ⊥; coord_id[p] ← −1;
state[p] ← undecided;
End Initialization

for c ← 1, 2, ..., N do // Process c becomes coordinator for four rounds
  1. Round 1:
     All undecided processes p send request(estimate[p], coord_id[p]) to c;
     if (c does not receive any request) then it skips rounds 2 to 4;
     else estimate[c] ← estimate[p] with largest coord_id[p];
  2. Round 2:
     c multicasts estimate[c];
     All undecided processes p that receive estimate[c] do
     estimate[p] ← estimate[c] and coord_id[p] ← c;
  3. Round 3:
     All undecided processes p that do not receive estimate[c] send NACK to c;
  4. Round 4:
     if (c does not receive any NACK) then c multicasts Decide; else c HALTS;
     All undecided processes p that receive Decide do
     decision[p] ← estimate[p];
     state[p] ← DECIDED;
end for

// prototype of a state update transition
if (round = 1) then
  1:  undC(⊥) ← undC(⊥) − 1;
  2:  req(⊥) ← req(⊥) + 1;
  3:  doneC(⊥) ← doneC(⊥) + 1;

The support variables doneC/F(m/⊥) are used to count the number of undecided processes that perform a transition, and to restore the appropriate initial values when switching to the successive round, with updates of the form undX(y) ← done.X(y), while the done variables are reset to 0. The round is incremented when all processes are moved to done, that is, when undC(m) + undF(m) + undC(⊥) + undF(⊥) = 0.

An interesting aspect is the modeling of send-omission failures. When using array theory, MCMT implicitly adopts the crash failure model [1], in that transitions describe the actions of alive processes. The behavior of faulty processes only omitting to send messages is fully described, whereas faulty crashed processes do nothing since their crashing (and if transitions do not cover all the possible cases, then the processes falling into the unspecified cases are considered crashing). Yet, in this case where the counter system is adopted, we have to maintain the accounting of all the processes. To this purpose, in every round we added transitions similar to the prototype above but without line 2: a faulty process may omit to send the request but it is counted among the triggered processes. Processes that crash are modeled as processes that from the failure on only perform those transitions.

Many kinds of transitions may be accelerated by changing the state of d processes (rather than 1) at a time, for any d greater than 0 and not greater than the global number of processes involved in the transition. In this case, the prototype above would be:

// prototype of a state update transition
if (round = 1 ∧ ∃d, 0 < d ≤ undC(⊥)) then
  1:  undC(⊥) ← undC(⊥) − d;
  2:  req(⊥) ← req(⊥) + d;
  3:  doneC(⊥) ← doneC(⊥) + d;

As an important heuristics, some model checkers (like MCMT) are supposed to produce by themselves the above accelerated form in order to prevent divergence and to speed up the verification for all possible values of the variable d in the indicated range, thus reproducing all cases of subsets of processes sending their requests and subsets of processes failing to do this. This acceleration does not introduce spurious traces.
Algorithm 2 Byzantine broadcast primitive

Round 1:
- **Phase 1:** process \( p \) sends \( \text{Init}(p, m, 1) \);
- **Phase 2:** \( \forall \) process does:
  - if (received \( \text{Init}(p, m, 1) \) from \( p \) in Phase 1) then send \( \text{Echo}(p, m, 1) \) to all;
  - if (received \( \text{Echo}(p, m, 1) \) from \( \geq N - t \) distinct processes in Phase 2) then \( \text{Accept}(p, m, 1) \);

Round 2:
- \( \forall \) Phase, \( \forall \) process does:
  - if (received \( \text{Echo}(p, m, 1) \) from \( \geq N - 2t \) distinct processes in previous phases \& not sent echo yet) then send \( \text{Echo}(p, m, 1) \) to all;
  - if (received \( \text{Echo}(p, m, 1) \) from \( \geq N - t \) distinct processes in this and previous phases) then \( \text{Accept}(p, m, 1) \);

A critical aspect of counter modelization is the **determination of the most recently distributed estimate** (lines 4, 8 of Algorithm 1). We describe the solution we adopted in our simulating model. If the coordinator receives just one type of requests (i.e., either \( \text{req}(m) = 0 \) or \( \text{req}(\perp) = 0 \)), then that value is taken as the estimate \( e \). Otherwise, two global variables \( \text{lastE} \) and \( \text{flagC} \) are used. The former is set to \( e \) every time the current coordinator succeeds in sending its estimate to at least one process. The latter is initialized to 0 and set to 1 the first time a coordinator reaches a correct process, which will report the estimate to all the successive coordinators. In the successive phases, if \( \text{flagC} > 0 \) then the coordinator takes \( \text{lastE} \) as its estimate; otherwise, the prover explores the algorithm behavior for both \( e = m \) and \( e = \perp \).

Differently from the array-based implementation, there is no memory about what processes are or have already taken the role as a coordinator: this is an example of an abstraction we are forced to make because of the adoption of counters formalization (this abstraction, fortunately, does not introduce spurious behavior). In Round 2, both cases of a correct and a faulty coordinator are explored, depending on whether the corresponding sets of processes are not empty. The two branches are distinguished by properly setting a global flag \text{correct coordinator}. In the case of a correct coordinator, all undecided processes adopt its estimate in one transition, and Round 3 is skipped, according to the following code (where \( e \) is supposed equal to \( m \)):

\[
\text{if } (e = m \land \text{correct coordinator}) \text{ then}
\quad \text{undC}(m) \leftarrow \text{undC}(m) + \text{undC}(\perp); \quad \text{undF}(m) \leftarrow \text{undF}(m) + \text{undF}(\perp);
\]

In the case of a faulty coordinator, transitions adopt the schema of the prototype above, so that processes are allowed to update their estimate – and their state and associated counter is changed accordingly – or alternatively to move to \( \text{done} \) thus simulating the send-omission failure of the coordinator. Moreover, at any time a transition allows to move to the successive round (describing the case where no further messages are sent by the coordinator). The residual number of undecided processes not yet triggered is the number of processes enabled to send a \text{nack}. This is modeled similarly to Round 1, with the simplification that it is sufficient one process sending the \text{nack} to switch to the successive round. Round 4 (diffusion of the \text{Decide} by the coordinator) is modeled after Round 2.

3.2 Byzantine broadcast primitive

In [22], an algorithm implementing a broadcast primitive for byzantine failures (BBP) is proposed, aiming at substituting authentication obtained by unforgeable signatures. Byzantine failures allow faulty processes to behave arbitrarily, i.e. omitting to send and/or receive messages, sending messages with a wrong content, or even coalescing to fool correct processes. The resilience in this case is \( t \leq (N - 1)/3 \). The algorithm attempts to achieve its goal through the re-diffusion of a message \( m \) sent by a source \( p \), on behalf of the receiving processes, trying to aggregate a majority of correct processes supporting the acceptance of \( m \). The pseudo-code is supplied by Algorithm 2. The algorithm aims at guaranteeing the following properties:

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

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

This algorithm significantly differs from those in sec.3.1. All algorithms are synchronous. Yet, in the previous algorithms, for each round a different kind of message is exchanged, and all messages of that type must arrive within that round. By contrast, in BBP just one type of message is exchanged, i.e. the Echo messages, and processes consider the cumulative number of Echo’s received so far. This is the reason why in our model we abstract from the round value, and just differentiate two phases: (i) initialization, depending on the property to be validated, and executed just once; (ii) actions undertaken by correct processes depending on the number of Echo’s each one of them received so far. The evolution of correct processes is reproduced by continuously triggering the transitions of phase (ii).

In our model, we consider different process states depending on the received messages. For correct processes, four states are possible: the *initial* state ($IT$), the *receivedInit* ($RI$) state, the *sentEcho* ($SE$) state and the *Accepted* ($AC$) state. Fig. 1 represents the state transitions: arcs are labeled with a pair (triggering event, performed action) – with one of the two elements possibly empty – and describe all the possible interleavings of events. In our counter specification, four global variables named after the states are used to count the number of processes in each state. According to Algorithm 2 and to the number of received Echo’s, correct processes move from one state to the other as shown in Fig. 1. These transitions are modeled simply by updating the number of processes in the four states, and accumulating the number of newly generated messages, which will be counted by the processes successively (phase (ii)).

The behavior of faulty processes is modeled indirectly through a global variable $F$ counting the number of Echo’s generated by faulty processes and received by the correct process taking the next transition. $F$ is updated by two transitions that may freely interleave with the others, which respectively increment or decrement $F$ while always guaranteeing that $0 \leq F \leq t$. The decrement is the *abstraction* we introduce to describe the possibility that a faulty process does not send its messages to all the correct processes (the abstraction consists in the fact that in this way we allow messages to be in a sense “withdrawn”). In our experiments, this abstraction does not introduce spurious behavior (for a different abstraction, using interval abstract domains, see [17]). It is worth to notice that correct processes are unable to distinguish whether an Echo was generated by either a correct or a faulty process. Hence, in the transitions modeling correct behavior, just the cumulative value ($SE + F$) is considered.

As far as the validation is concerned, we produced one model for every property to be validated: all models include the same transitions and they differ just in the initialization. Unforgeability is a *safety* property, while both Correctness and Relay are *liveness* properties. Yet, most of the available infinite-state model checkers are not able to verify liveness properties. In order to deal with this problem, we change liveness properties into safety properties by exploiting the round indication in the assertion. In the following, for each property we supply both the corresponding unsafe formula, and the initial state:

- **Correctness:** the unsafe formula is: $\varphi_C := ((SE + F) < (N - t) \land (IT + RI + AC) = 0)$. Initially all correct processes are in $RI$ state.
- **Unforgeability:** the unsafe formula is: $\varphi_U := (AC > 0)$. Initially all correct processes are in $IT$ state.
- **Relay:** to refute acceptance on behalf of the correct processes, it must be the case that either (i) there are still processes in $IT$ but the Echo messages are not sufficient to move them to either $SE$ or
AC state; or (ii) there are still processes in SE state but the Echo messages are not enough to move them to AC state. The two conditions above are described by the following unsafe formulas, which are checked separately:

\[ \varphi_{R_A} := ((SE + F) < (t + 1) \land IT > 0 \land RI = 0) \]
\[ \varphi_{R_B} := (AC < (N - t) \land (IT + RI) = 0 \land (SE + F) < (N - t)) \]

For both Relay\_A and Relay\_B, initially the number of Echo’s sent by correct processes equals SE + AC and processes may be in whatever of the four states, but it must be AC > 0.

# 4 Performance Evaluation

We implemented the models described in the previous section in MCMT, in order to validate the algorithms and to measure the performance obtained under different modeling paradigms. The MCMT models have been automatically translated into equivalent formalizations in either µZ and nuXmv, in order to gain some understanding of the behavior of the analyzed approaches. The translators we developed yield a twofold advantage: they allow to model an algorithm only once, and they allow to perform a fair comparison amongst tools. For the CBS and SOBA algorithms, the results are also compared with a modelization using array-based systems, described in [1]. Array modelizations of the correctness, unforgeability and relay properties have been developed for this work. It should be noticed that array modelizations do not come from automatic translations of the corresponding counters specifications, so that they must be considered as related but different benchmarks. Another important point is that in the case of array modelizations, convergence of the computation is often (although not always) obtained by manual manipulation of some formal abstraction parameters (see the MCMT User Manual included in the distribution\(^2\)). By contrast, convergence for the counter-specification models has been achieved in the default configurations simply by acceleration (in the case of MCMT) or by IC3-like abstraction/refinement (in the case of µZ and nuXmv).

All measures have been conducted on an Intel Core i5-2500 CPU @ 3.30GHz with 8 GB RAM, running Debian GNU/Linux stretch/sid x86\_64, and with MCMT version 2.5.2 and Yices version 1.0.40.

Table 1 reports the obtained results. The File column contains the name of the model file – which identifies the considered algorithm and property to be validated – and the failure model in which the validation was performed. The Result column reports the outputs of the model checkers: for all tests all the model checkers supplied the same output. For MCMT, we also report the maximum Depth of the explored tree, and the number of Nodes (system states) composing the explored tree, as an indication of the computation overhead in the various cases.

The results show that all the considered tools solve the algorithms efficiently. MCMT is not as engineered as the other two tools, but its performance is nonetheless comparable although not as good. Results lead to the conclusions that pure SMT is a technology suitable to perform the automatic verification of fault-tolerant distributed algorithms, without the need of transforming those systems into finite-state systems. Array-based modeling is usually computationally heavier, which might be expected as it uses a more expressive logic. Counter systems are lighter, but they require a significant amount of a-priori work in order to infer key points of the algorithm behavior from its specification and include them in the model. An example of this is the mechanism used to model the choice of the more recently distributed estimate in SOBA.

One word about heuristics needed to prevent divergence: as we already mentioned, MCMT can only use acceleration for counters systems, whereas µZ and nuXmv use sophisticated forms of abstraction/refinement. Since acceleration is just a preprocessing technique, one can manually include accelerated transitions in the specification files for µZ and nuXmv too; however, we did not notice significant improvement doing that in the above benchmarks (but the overhead is also modest). On the other hand, the most difficult benchmark (namely the counters formalization of GOBA) has been solved only by MCMT, thus showing that plain acceleration without abstraction may also be the winning choice in some cases. Finally, we mention that the array based specification of GOBA has not been verified yet.

\(^2\)http://users.mat.unimi.it/users/ghilardi/mcmt/
### Table 1: Performance evaluation

<table>
<thead>
<tr>
<th>File</th>
<th>Result</th>
<th>MCMT</th>
<th>$\mu Z$</th>
<th>mjXmv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Depth</td>
<td>Nodes</td>
<td>Time</td>
</tr>
<tr>
<td>crash.mcmt (crash)</td>
<td>SAFE</td>
<td>8</td>
<td>39</td>
<td>0.38 s.</td>
</tr>
<tr>
<td>crash.mcmt (send-om.)</td>
<td>UNSAFE</td>
<td>6</td>
<td>33</td>
<td>0.14 s.</td>
</tr>
<tr>
<td>sendom.mcmt (send-om.)</td>
<td>SAFE</td>
<td>21</td>
<td>17722</td>
<td>77 s.</td>
</tr>
<tr>
<td>sendom.mcmt (general-om.)</td>
<td>UNSAFE</td>
<td>8</td>
<td>76</td>
<td>0.28 s.</td>
</tr>
<tr>
<td>generalF.mcmt (general-om.)</td>
<td>SAFE</td>
<td>42</td>
<td>10102</td>
<td>77 s.</td>
</tr>
<tr>
<td>byz.forg.mcmt (byzantine)</td>
<td>SAFE</td>
<td>7</td>
<td>51</td>
<td>0.48 s.</td>
</tr>
<tr>
<td>byz.forg.mcmt (t &gt; (N−1)/3)</td>
<td>UNSAFE</td>
<td>4</td>
<td>24</td>
<td>0.10 s.</td>
</tr>
<tr>
<td>byz.corr.mcmt (byzantine)</td>
<td>SAFE</td>
<td>7</td>
<td>131</td>
<td>2.55 s.</td>
</tr>
<tr>
<td>byz.corr.mcmt (t &gt; (N−1)/3)</td>
<td>UNSAFE</td>
<td>4</td>
<td>37</td>
<td>0.24 s.</td>
</tr>
<tr>
<td>byz.relayA.mcmt (byzantine)</td>
<td>SAFE</td>
<td>6</td>
<td>38</td>
<td>0.22 s.</td>
</tr>
<tr>
<td>byz.relayA.mcmt (t &gt; (N−1)/3)</td>
<td>UNSAFE</td>
<td>1</td>
<td>2</td>
<td>0.01 s.</td>
</tr>
<tr>
<td>byz.relayB.mcmt (byzantine)</td>
<td>SAFE</td>
<td>8</td>
<td>185</td>
<td>4.97 s.</td>
</tr>
<tr>
<td>byz.relayB.mcmt (t &gt; (N−1)/3)</td>
<td>UNSAFE</td>
<td>1</td>
<td>2</td>
<td>0.02 s.</td>
</tr>
<tr>
<td>crash_array.mcmt (crash) [1]</td>
<td>SAFE</td>
<td>13</td>
<td>113</td>
<td>0.44 s.</td>
</tr>
<tr>
<td>crash_array.mcmt (send-om.) [1]</td>
<td>UNSAFE</td>
<td>12</td>
<td>464</td>
<td>17.66 s.</td>
</tr>
<tr>
<td>sendom_array.mcmt (send-om.) [1]</td>
<td>SAFE</td>
<td>39</td>
<td>11206</td>
<td>2317 s.</td>
</tr>
<tr>
<td>unforg_array.mcmt (byzantine)</td>
<td>SAFE</td>
<td>17</td>
<td>368</td>
<td>2.07 s.</td>
</tr>
<tr>
<td>corr_array.mcmt (byzantine)</td>
<td>SAFE</td>
<td>3</td>
<td>16</td>
<td>0.03 s.</td>
</tr>
<tr>
<td>relay1_array.mcmt (byzantine)</td>
<td>SAFE</td>
<td>2</td>
<td>26</td>
<td>0.04 s.</td>
</tr>
<tr>
<td>relay2_array.mcmt (byzantine)</td>
<td>SAFE</td>
<td>4</td>
<td>21</td>
<td>0.03 s.</td>
</tr>
</tbody>
</table>

### 5 Conclusions and Related Work

In this paper, we showed how Satisfiability Modulo Theories may be effectively used to perform automated formal verification of fault-tolerant distributed algorithms. Algorithms for the Reliable Broadcast problem have been modeled in MCMT using both the array and the counter specification paradigm. To the best of our knowledge, this is the first modelization of algorithms for crash, send-omission and general omission failures using this paradigm. In all cases, the verification converged. We reported some performance comparison with other SMT-based model checkers operating on equivalent models, and with the modelization of the same algorithms using the array-based paradigm. In all cases, promising performances have been obtained, showing that SMT is a formalism powerful enough to manipulate this sort of problems.

A series of papers related to ours is [17, 19, 20]; in these papers, abstractions leading to counters formalizations are introduced for a large set of fault-tolerant distributed algorithms (including those from Subsection 3.2 above). An interesting specification formalism, namely *threshold automata*, is introduced. Roughly speaking, threshold automata are counter automata in which integer variables are divided into two groups: the variables in the first group measure the number of processes in each location, whereas the variables in the second group (called ‘shared variables’) measure the progress of the system and, as such, cannot be decremented. It is assumed that in each cycle of the control flow of the automaton, shared variables cannot increase either; under these assumptions, it is proved in [19] that the system diameter is finite, thus allowing bounded model checking to be complete for verification. Optimizations in the trace enumerations are designed in [20]; once relevant traces are identified, the whole verification task can be discharged by an SMT-solver checking the actual feasibility of such traces. Overall, this approach seems to be quite powerful and scalable, whenever abstractions needed for encoding algorithms and problems into the proposed formalism are available (it is not clear for instance, whether our examples from Subsection 3.1 can fit this framework).

As a future work, we plan to model the whole algorithm for Reliable Broadcast in the presence of byzantine failures. We will extend the logic mechanisms used for reasoning on array-based systems so as to be able to count the number of processes in a certain state or, more generally, fulfilling a certain property; this enrichment might be able to capture some advantages of counter formalizations inside the framework of array-based systems.

The considered algorithms work for certain values of resilience, which have been inferred in the literature only through informal, verbal proofs. Another interesting aspect would be to use model checkers in order...
to automatically discover resilience thresholds in the verification process, as those thresholds that separate safe and unsafe executions.

Acknowledgments
The second author would like to acknowledge the support of the PRIN 2010-2011 project ‘Logical Methods for Information Management’ funded by the Italian Ministry of Education, University and Research (MIUR).

References


