Counter Abstractions in Model Checking of Distributed Broadcast Algorithms: some case studies

Francesco Alberti[1], Silvio Ghilardi[2], Andrea Orsini[2], Elena Pagani[2]

[1] Fondazione Centro San Raffaele, Milano, Italy
[2] Università degli Studi di Milano, Milano, Italy

Abstract. The automated, formal verification of distributed algorithms is a crucial, although challenging, task. 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*. In fact, this simulation does not indeed introduce spurious runs violating the safety properties we want to formally verify in a significant class of problems.
We report the related performance evaluations of some SMT-based model-checkers (essentially, our tool MCMT and tools like μZ, 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.

1 Introduction

The validation of fault-tolerant distributed algorithms like those in [7, 25] 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 characteristics 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., [24]) 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.

In our opinion, *array-based systems* [17] offer the most suitable formal model for these verification tasks; however, only quite recently decision procedures for arrays with cardinality constraints were designed and implemented [1] (see also [14] for a related framework) and a model checker exploiting them in a fully automated setting is still under construction. Our current tool MCMT - 'Model Checker Modulo Theories' [18] implements array-based specifications, but the impossibility of expressing in a direct way cardinality constraints causes difficulties in handling verification of fault-tolerant distributed algorithms whenever threshold guards are involved (although sometimes ad hoc abstraction strategies may be successfully used).

In this paper, we report some experiments using *counter systems* [13] to specify the problems we consider. 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 often 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 well and is much more supported from the existing technology.* 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 [2]. Finally, in Section 4 we report a performance evaluation of the verification with different state-of-the-art SMT-based tools.

2 Preliminaries

The behavior of a reactive system can be modeled through a *transition system*, which is a triple $\mathcal{T} = (W, W_0, R)$ 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: $w_1 R w_2$ 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

$$w_o R w_1 R w_2 \cdots R w_n$$

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 $w\rho w'$; (ii) if $w\rho w'$ and $w \in W_0$ then $w' \in W_0'$; (iii) if $w\rho w'$ and $w \in Bad$ then $w' \in Bad'$; (iv) if $w\rho w'$ and wRv there is a path $w'R'w_1' \cdots w_n'R'v'$ with $v\rho v'$. It is evident that the existence of such a simulation ρ guarantees that if there is no path in (W', W_0', R') leading from W_0' to Bad', then there is no path in (W, W_0, R) leading from W_0 to Bad. Thus one can model-check (W', W_0', R') (w.r.t. Bad') instead of (W, W_0, R) (w.r.t. 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 Bad' 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 more or less straightforward details. It is not clear whether such details can be checked automatically (they are specific to each example); this might be seen a weak point of the method, however it should be pointed out that the substantial content of the verification task is in fact lifted to the simulating model, where it is checked automatically.

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 exaustive search through the states (in the style of textbooks like [9]) is not possible; states must be handled *symbolically* through logical formulae. In addition, satisfiability tests for these formulae 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, an SMT-solver (Z3 [11], Yices [15], MathSAT [8], CVC4 [4],...) 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 formulae 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 $\underline{x}$, the initial and unsafe configurations are specified via formulae $W_0(\underline{x})$, $Bad(\underline{x})$ and the transition relation is specified via a formula $R(\underline{x}, \underline{x}')$. When this formal framework is fixed, the SMT-solver can be used for instance as follows. If we are supplied an invariant $I(\underline{x})$, the SMT-solver can check whether $Bad(\underline{x}) \wedge I(\underline{x})$ is satisfiable, thus proving in the negative case that states in 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 formulae

$$W_0(\underline{x}) \wedge \neg I(\underline{x}) \qquad\qquad I(\underline{x}) \wedge R(\underline{x}, \underline{x}') \wedge \neg I(\underline{x}') \ .$$

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 modelisation problem relying on the user's choices: we discuss our own choices in next subsection.

2.1 Choosing a formalization

We discuss two methods for describing distributed algorithms. First, we have the *array specification* relying on the notion of an *array-based* system [17]. 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 [18] and Cubicle [10]. Relevant properties need quantifiers to be expressed: for instance, the violation of mutual exclusion is expressed as

$$\exists z_1 \exists z_2 \ (z_1 \neq z_2 \wedge a[z_1] = \texttt{critical} \wedge a[z_2] = \texttt{critical}) \qquad (1)$$

where a maps every process to its location.

Counter systems is the 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 a process in a reactive parameterized system is driven by conditions of the type:

```
if (number of received messages is in [min, max])
    then perform action;
```

where *action* can be a location change, a message 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* [20, 21]). Counter specifications have been used to formalize broadcast protocols [12, 16], cache coherence protocols [13] and are the basic formalism accepted by automata-based tools like FAST [3]. 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 corrupted messages only to a subset of processes in order to fool them. 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 [20] 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 [1] for recent promising results in this direction).

2.2 The tools

Whereas for array specifications the Model Checker Modulo Theories (MCMT) [18] is our reference tool, for counter specifications, it is possible to use also other SMT-based model-checkers (we choose μZ [19] and nuXmv [6]) because counter specifications only involve integer variables. In detail:

- MCMT is an SMT-based model checker able to verify the properties of infinite-state reactive parameterized systems. Although specifically designed for array-

[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).

based systems, it can handle counter specifications too. MCMT exploits Yices [15] as the underlying SMT-solver. 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. On the other hand, MCMT implements also some form of *acceleration*; by using acceleration the tool may become competitive for counter specifications, because it is able to describe in one shot the effect of executing some loops any finite number of times.

- μZ 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 problems addressed in the counter specifications of this paper. As a search mechanism, μZ employs an IC3-based algorithm - a sophisticated abstraction/refinement search driven by the property to be checked.
- nuXmv 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 [7], and the algorithm for a broadcast primitive with byzantine failures (BBP) described in [25]. 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.

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

Algorithm 1 Pseudo-code for CBA and SOBA

```
Initialization:
    if ($p$ is the sender)
        then $estimate[p] \leftarrow m$; $\underline{coord\_id[p] \leftarrow 0;}$
        else $estimate[p] \leftarrow \perp$; $\underline{coord\_id[p] \leftarrow -1;}$
    $state[p] \leftarrow undecided$;
End Initialization

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

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 [2], 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 ($undC/F(m/\perp)$),[2] (*ii*) the number of either correct or faulty processes that have decided ($decC/F(m/\perp)$), (*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 $(undC(m) + undF(m)) = 1$. The *unsafe formula* to be refuted, negating the Agreement property, is expressed as: $\varphi = (decC(m) > 0 \wedge decC(\perp) > 0)$.

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 enabling the transition of undecided correct processes with no message is:

```
// prototype of a state update transition
if ($round = 1$) then
        1: $undC(\perp) \leftarrow undC(\perp) - 1$;
        2: $req(\perp) \leftarrow req(\perp) + 1$;
        3: $doneC(\perp) \leftarrow doneC(\perp) + 1$;
```

The support variables $doneC/F(m/\perp)$ 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 $undX(y) \leftarrow doneX(y)$, while the *done* variables are reset to 0. The *round* is incremented when all processes are moved to *done*, that is, when $undC(m) + undF(m) + undC(\perp) + undF(\perp) = 0$.

An interesting aspect is the **modeling of send-omission failures**. When using array theory, MCMT implicitly adopts the crash failure model [2], 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 crashed). The above solution is based on an (implicit) quantifiers relativization technology, which is not available anymore when counter abstractions are adopted; so in counter systems, 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 an accelerated state update transition
if ($round = 1 \wedge \exists d, 0 < d \leq undC(\perp)$) then
        1: $undC(\perp) \leftarrow undC(\perp) - d$;
```

[2] Thus e.g. $undC(m)$ counts the number of undecided correct processes owning m, whereas $undF(\perp)$ counts the number of undecided faulty processes not owning m, etc.

```
2: $req(\bot) \leftarrow req(\bot) + d$;
3: $doneC(\bot) \leftarrow doneC(\bot) + d$;
```

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.*

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 $req(m) = 0$ or $req(\bot) = 0$), then that value is taken as the estimate e. Otherwise, two global variables *lastE* and *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 $flagC > 0$ then the coordinator takes *lastE* as its estimate; otherwise, the algorithm behavior is nondeterministic: both $e = m$ and $e = \bot$ are allowed.

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 `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):

```
if ($e = m$ ∧ correct_coordinator) then
    $undC(m) \leftarrow undC(m) + undC(\bot)$;  $undF(m) \leftarrow undF(m) + undF(\bot)$;
```

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 *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 NACK. This is modeled similarly to Round 1, with the simplification that it is sufficient one process sending the NACK to switch to the successive round. Round 4 (diffusion of the *Decide* by the coordinator) is modeled after Round 2.

3.2 Byzantine broadcast primitive

In [25], an algorithm implementing a broadcast primitive for byzantine failures (BBP) is proposed, aiming at substituting authentication obtained by unforge-

Algorithm 2 Byzantine broadcast primitive

```
Initialization: $k \leftarrow$ round number when $p$ broadcasts $m$ ;
Round k:
    Phase (2k-1): process $p$ sends Init($p$,$m$,$k$);
    Phase 2k: $\forall$ process does:
        if (received Init($p$, $m$, $k$) from $p$ in Phase $2k-1$) then send Echo($p$, $m$, $k$) to all;
        if (received Echo($p$, $m$, $k$) from $\geq N-t$ distinct processes in Phase $2k$)
          then Accept($p$, $m$, $k$);
Round $r \geq$ k+1:
    $\forall$ Phase $(2r-1)$, $2r$: $\forall$ process does
        if (received Echo($p$, $m$, $k$) from $\geq N-2t$ distinct processes in previous phases
           $\wedge$ not sent echo yet) then send Echo($p$, $m$, $k$) to all;
        if (received Echo($p$, $m$, $k$) from $\geq N-t$ distinct processes in this and previous
              phases) then Accept($p$, $m$, $k$);
```

able 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 a 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

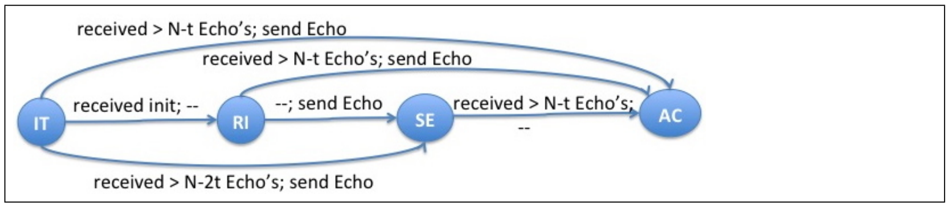
received > N-t Echo's; send Echo
received > N-t Echo's; send Echo
received init; --
--; send Echo
received > N-t Echo's;
--
IT
RI
SE
AC
received > N-2t Echo's; send Echo

Fig. 1: State transitions for a correct process in BBP.

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 [20]). 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: since the round where the 'good event' (i.e. the event mentioned in the liveness property) has to take place is known, the problem can be reformulated as a safety property (even better, as a bounded model checking property). In the following, for each property we supply both the corresponding unsafe formula, and the initial state:

– **Correctness:** the unsafe formula is: $\varphi_C := ((SE{+}F) < (N{-}t) \wedge (IT{+}RI) = 0)$. 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:

$$\varphi_{R_A} := ((SE + F) < (t + 1) \wedge IT > 0 \wedge RI = 0)$$

$$\varphi_{R_B} := (AC < (N - t) \wedge (IT + RI) = 0 \wedge (SE + F) < (N - t)).$$

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; 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. It should be noticed that verifying counter specifications is undecidable in general, thus there is no guarantee that our tools converge. To achieve convergence in practical cases, one needs sophisticated methods: these include acceleration (in the case of MCMT) or 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. The experiments have been conducted with MCMT version 2.5.2 leveraging Yices 1.0.40, μZ version 4.4.2, and nuXmv version 1.0.1.[3]

Table 1 reports the results obtained with the best combination of parameters for each tool, which we supply in our website. The *Model* column 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* (formulae describing 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*

[3]After the submission of this paper, a new version 1.1.0 of nuXmv was released; we repeated the experiments with the new release without noticing substantial differences.

Model	Result	MCMT			μZ	nuXmv
		Depth	Nodes	Time	Time	Time
CBA - Agreement (crash failures)	SAFE	8	39	0.38 s.	0.66 s.	0.41 s.
CBA - Agreement (send-om.)	UNSAFE	6	33	0.14 s.	0.14 s.	0.16 s.
SOBA - Agreement (send-om.)	SAFE	21	1772	77 s.	628 s.	19 s.
SOBA - Agreement (general-om.)	UNSAFE	8	76	0.28 s.	0.26 s.	0.48 s.
GOBA - Agreement (general-om.)	SAFE	42	10102	6308 s.	> 24 h.	> 24 h.
GOBA - Agreement ($t > (N-1)/2$)	UNSAFE	22	6953	2523 s.	953 s.	25 s.
BBP - Unforgeability (byzantine)	SAFE	7	51	0.48 s.	0.20 s.	0.03 s.
BBP - Unforgeability ($t > (N-1)/3$)	UNSAFE	4	24	0.10 s.	0.05 s.	0.14 s.
BBP - Correctness (byzantine)	SAFE	7	131	2.55 s.	2.54 s.	0.21 s.
BBP - Correctness ($t > (N-1)/3$)	UNSAFE	4	37	0.24 s.	0.10 s.	0.30 s.
BBP - Relay$_A$ (byzantine)	SAFE	6	38	0.22 s.	0.04 s.	0.08 s.
BBP - Relay$_A$ ($t > (N-1)/3$)	UNSAFE	1	2	0.01 s.	0.03 s.	0.12 s.
BBP - Relay$_B$ (byzantine)	SAFE	8	185	4.97 s.	0.08 s.	0.17 s.
BBP - Relay$_B$ ($t > (N-1)/3$)	UNSAFE	1	2	0.02 s.	0.05 s.	0.11 s.

Table 1. Performance evaluation

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.

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.

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 using the counter specification paradigm. This paradigm is well-known [13], but 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 and 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 [20, 22, 23]; 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 [22] that the system diameter is finite, thus allowing bounded model checking to be complete for verification. Optimizations in the trace enumerations are designed in [23]; 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. The idea is to come back to array-based systems like in [2], but using recent achievements from [1] in order to be able to capture some advantages of counter formalizations inside the framework of array-based systems. The challenge is that of keeping the efficiency of counter systems documented in this paper in a more expressive and more flexible context, allowing direct formalizations and avoiding ad hoc simulations/abstractions. This would be in some sense similar to the approach taken in the forthcoming paper [5], where array specifications with cardinality constraints are joined to the machinery of Horn clauses solving (the difference is that we plan to use complete [1] or at least more aggressive decision procedures for generating and discharging proof obbligations involving counting quantifiers fragments).

The algorithms considered in this paper 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.

References

1. F. Alberti, S. Ghilardi, and E. Pagani. Counting Constraints in Flat Array Fragments. In *Proc. IJCAR*, Lecture Notes in Computer Science, 2016. Preliminary version in arXiv:1602.00458.
2. F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G.P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 8:29–61, 2012.
3. S. Bardin, J. Leroux, and G. Point. Fast extended release. In *Proc. of CAV*, pages 63–66, 2006.
4. C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. Int'l Conf. on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

5. N. Bjørner, K. von Gleissenthall, and A. Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *Proc. of the 37th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2016.
6. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In *Proc. 26th Int'l Conf. on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, Jul. 2014. `https://nuxmv.fbk.eu/`.
7. T.D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Proc. 4th Int'l Workshop on Distributed Algorithms*, volume 486 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 1991.
8. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *Lecture Notes in Computer Science*. Springer, 2013.
9. E. M. Clarke, O. Grunberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
10. S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In *Proc. 24th Int'l Conf. on Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, pages 718–724. Springer, Jul. 2012.
11. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, Mar. 2008. `https://github.com/Z3Prover/z3`.
12. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. In *Proc. of CSL*, volume 1683 of *LNCS*, pages 50–66, 1999.
13. Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
14. C. Dragoj, T. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In *Proc. of VMCAI*, 2014.
15. B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006. `http://yices.csl.sri.com`.
16. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. of LICS*, pages 352–359. IEEE Computer Society, 1999.
17. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model Checking of Array-based Systems. In *Proc. Int'l Joint Conf. on Automated Reasoning (IJCAR)*, volume 5195 of *Lecture Notes in Computer Science*, pages 67–82. Springer, Aug. 2008.
18. S. Ghilardi and S. Ranise. MCMT: a Model Checker Modulo Theories. In *Proc. Int'l Joint Conf. on Automated Reasoning (IJCAR)*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, Jul. 2010. `http://users.mat.unimi.it/users/ghilardi/mcmt/`.
19. K. Hoder, N. Bjørner, and L. De Moura. μZ - an efficient engine for fixed points with constraints. In *Proc. of CAV*, LNCS, 2011.
20. A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 201–209, Aug. 2013.

21. A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms. In *Proc. Int'l SPIN Symposium on Model Checking of Software*, volume 7976 of *Lecture Notes in Computer Science*, pages 209–226. Springer, Jul. 2013.
22. I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In *Proc. of CONCUR*, LNCS, pages 125–140, 2014.
23. I. Konnov, H. Veith, and J. Widder. SMT and POR beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms. In *Proc. of CAV*, LNCS, 2015.
24. M.C. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
25. T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.