What You Always Wanted to Know about Model Checking of Fault-Tolerant Distributed Algorithms*

Igor Konnov, Helmut Veith, and Josef Widder

TU Wien (Vienna University of Technology)

Abstract. Distributed algorithms have numerous mission-critical applications in embedded avionic and automotive systems, cloud computing, computer networks, hardware design, and the internet of things. Although distributed algorithms exhibit complex interactions with their computing environment and are difficult to understand for human engineers, computer science has developed only very limited tool support to catch logical errors in distributed algorithms at design time.
In the last two decades we have witnessed a revolutionary progress in software model checking due to the development of powerful techniques such as abstract model checking, SMT solving, and partial order reduction. Still, model checking of fault-tolerant distributed algorithms poses multiple research challenges, most notably parameterized verification: verifying an algorithm for all system sizes and different combinations of faults. In this paper, we survey our recent results in this area which extend and combine abstraction, partial orders, and bounded model checking. Our results demonstrate that model checking has acquired sufficient critical mass to build the theory and the practical tools for the formal verification of large classes of distributed algorithms.

1 Introduction

Fault-tolerant distributed algorithms (FTDA) are a central research area in distributed computing theory [28,2]. While such algorithms typically have been used in safety critical applications in the automotive or avionic industries, new application domains such as cloud computing provide additional motivation to study fault-tolerant algorithms: with the huge number of computers involved in a cloud, faults are the norm [30] rather than an exception. Together, this motivates our research on automated verification techniques for fault-tolerant distributed algorithms. We need to automatically verify such mechanisms for several hundreds or even thousands of components. However, a straightforward application of model checking to systems of such a scale suffers from combinatorial state space explosion.

* Supported by the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403 and S11405) and project P27722 (PRAVDA), and by the Vienna Science and Technology Fund (WWTF) through project ICT15-103 (APALACHE).

A paradigmatic approach to verify very large systems is *parameterized model checking*: if $M(n)$ is a distributed or concurrent system consisting of n identical components, and ϕ is a temporal logic formula, parameterized model checking requires us to check whether $\forall n.\, M(n) \models \phi$. Already for quite restricted classes of concurrent systems the problem is undecidable, cf. our recent survey [4]. For fault-tolerant distributed algorithms there are (at least) two more challenges that we shall discuss below: (i) multiple parameters with arithmetic constraints and (ii) parameterized code. Let us describe these challenges more precisely. First, in addition to n there is a parameter t that expresses the assumed number of faulty components, and algorithms are typically correct only under a *resilience condition*. A typical resilience condition in the case of Byzantine fault tolerance [31,14] is $n > 3t$. Second, while the parameterized model checking problems discussed in [4] assume that the process code and state space are independent of the parameters, FTDAs often count messages: Due to faults, processes cannot wait for messages from specific (possibly faulty) senders. Therefore, most FTDAs use counters, e.g., if a process receives a certain message from more than t distinct senders, then it concludes that one of the senders must be non-faulty. We call such conditions on counters *threshold guards*.

Algorithm 1 Core logic of the broadcasting algorithm from [35].

Code for processes i if it is correct:

Variables

1: $v_i \in \{\text{FALSE}, \text{TRUE}\}$
2: $\text{accept}_i \in \{\text{FALSE}, \text{TRUE}\} \leftarrow \text{FALSE}$

Rules

3: **if** v_i **and** not sent $\langle$echo$\rangle$ before **then**
4: *send* $\langle$echo$\rangle$ to all;
5: **if** *received* $\langle$echo$\rangle$ from at least $t{+}1$ *distinct* processes **and** not sent $\langle$echo$\rangle$ before **then**
6: *send* $\langle$echo$\rangle$ to all;
7: **if** *received* $\langle$echo$\rangle$ from at least $n - t$ *distinct* processes **then**
8: $\text{accept}_i \leftarrow \text{TRUE}$;

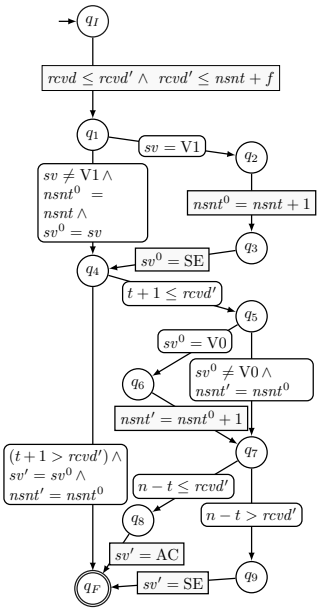


Fig. 1. A control flow automaton of the Algorithm 1 for Byzantine faults.

Algorithm 1 presents a threshold-based FTDA in pseudo code, as is typical for the distributed algorithms literature. It uses threshold guards in lines 5 and 7. In Figure 1, we give a graphical representation of a control-flow automaton that serves as a formal representation of the algorithm. For instance, the local variable *rcvd* represents the number of received messages, which is implicitly assumed in the pseudo code, while the global variable *nsnt* represents the number of messages sent by the correct processes. Moreover, the local variable *sv* represents the local control state of a process, which is implicit in the pseudo code in the phrases "not sent $\langle echo \rangle$ before" and "$accept_i \leftarrow$ TRUE". Note that the expressions over the parameters are compared to the value of variable $rcvd'$, which contains the number of received messages, including the messages received at the current step. A system is then composed of $n - f$ instances of the control-flow automaton that run concurrently and represent the correct processes. The formal definition and the semantics of control-flow automata can be found in [21].

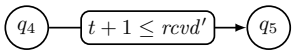
We observe that the process code and state space depend on the parameters (in our example on n and t). In addition to the parameterized number of processes and faults, automatic verification of FTDAs has to deal with process code which refers to parameters in a non-trivial way. We address this problem by stacking different techniques that we will survey in the following section.

2 Verification techniques

Figure 2 gives an overview of our techniques that we introduced in a series of papers on parametrized model checking of FTDAs [21,23,24]. In Section 3, we discuss how these techniques interact with each other in the framework of our tool ByMC. We deal with the parametrized code and state space by a parametrized interval data abstraction [21] in Section 2.1. After that step, we have obtained a more classic parametrized model checking problem where all processes are uniform [4] and the system is thus *symmetric*. Symmetry allows us to change representation into a counter representation (Section 2.2) which gives rise to different techniques, namely, counter abstraction (Sections 2.3 and 2.4), and offline partial order reduction with acceleration (Sections 2.5 to 2.7).

2.1 Parametric Interval Data Abstraction (PIA data)

In [21] we formalized threshold-guarded statements (e.g., the one from line 5 in the pseudocode example given in Algorithm 1) using a special form of control flow automata, e.g.:



The above edge from q_4 to q_5 can be executed only if the number of received messages $rcvd'$ is greater than or equal to $t + 1$. The central insight is that for evaluating this condition, the precise value of $rcvd'$ is not important, it suffices to know whether $rcvd'$ is above the threshold. Our case study in [21] contained

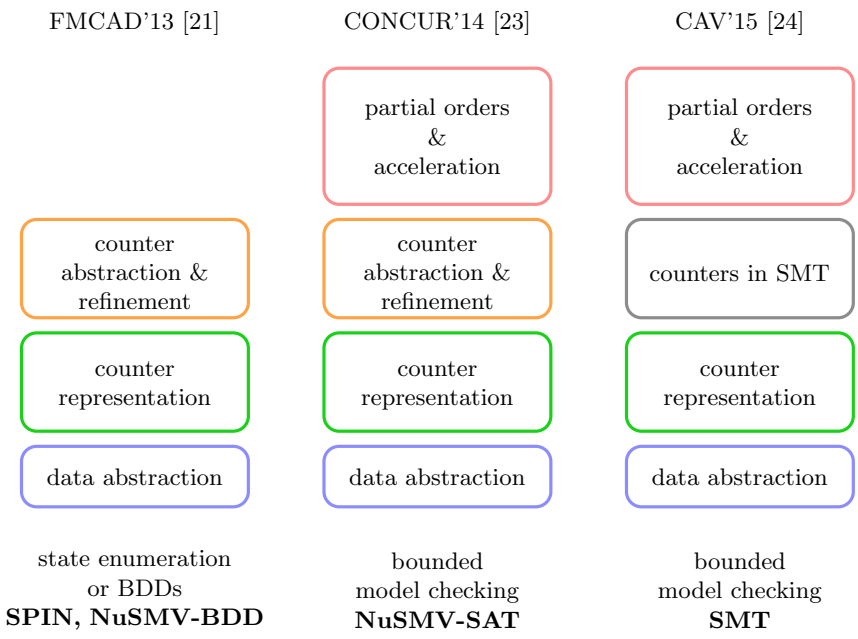
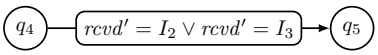


Fig. 2. Stacks of techniques

an additional threshold guard of $n-t$. This motivated an *abstract domain* of four intervals $I_0 = [0, 1[$ and $I_1 = [1, t+1[$ and $I_2 = [t+1, n-t[$ and $I_3 = [n-t, n]$. In our approach, the abstract domain is extracted from the guards automatically.

Recall that we want to get rid of parameterized process code. To this end, we can now replace the guards that refer to unbounded variables and parameters by their *abstraction*. In our abstract domain, if the guard "$t+1 \leq rcvd'$" evaluates to true, this means that $rcvd'$ is in the interval $[t+1, n-t[$ or $[n-t, n]$. These intervals correspond to the abstract values I_2 and I_3, respectively. Thus, we can replace the guard by:



In this way we obtain a finite-state abstract process. Still, the resulting system is a parallel composition of a parametric number of such processes.

2.2 Counter Representation

A system that consists of concurrent anonymous (identical) processes can be modeled as a counter system by exploiting the symmetry of the system: *Instead of recording which process is in which local state, we record for each local state, how many processes are in this state.* Thus, we need one counter per local state ℓ,

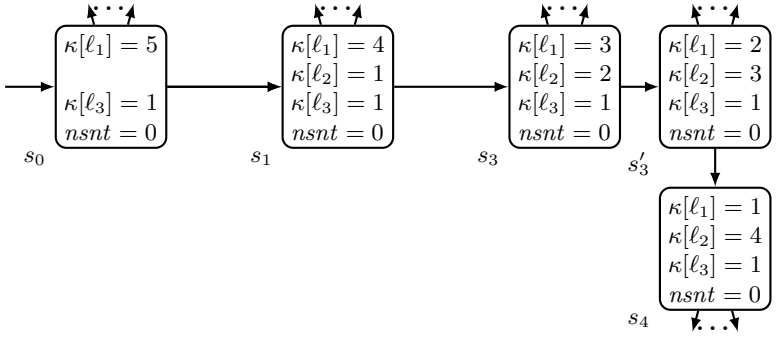


Fig. 3. An illustration of a counter representation for a system with $n = 7, t = 1, f = 1$. States s_3 and s'_3 correspond to the single abstract state $\hat{s}_3$ in Figure 4.

which we denote by $\kappa[\ell]$. After the PIA data abstraction, abstract processes have a fixed finite number of local states, hence we have a fixed number of counters. A step by a process that goes from local state ℓ to local state ℓ' is modeled by decrementing $\kappa[\ell]$ and incrementing $\kappa[\ell']$. When we fix the number of processes, e.g., by giving a concrete value to n, each counter is bounded by the number of processes n.

Figure 3 illustrates a transition system obtained by switching to a counter representation of a system of six correct processes (hence, the sum of counters is six in each state). Note that each transition decrements one counter and increments another one. As one can see, if the original system does not have self-loops, the counter representation does not have them either. This is in sharp contrast to counter abstraction, which is presented in Section 2.3.

However, as we are interested in the parametrized problem, we have to consider systems for all values of n. That is, after changing the representation, we have not reached a finite state representation. Thus another abstraction is needed.

Remark. In the literature, "counter representation" is sometimes referred to as "counter abstraction," partly because such a system can be viewed as more abstract due to absence of process identifiers. As the specifications of FTDAs do not single out processes but refer to process states only using quantification over the individual processes, for us this "counter representation" maintains all information which is present in the parallel composition of processes. Thus, in our setting, the counter representation is precise for the specifications of FTDAs that quantify over all correct processes.

2.3 Parametric Interval Counter Abstraction (PIA counter)

In the counter representation of Section 2.2, the unbounded counter values are the only source of an unbounded state space. To get rid of this, the natural idea

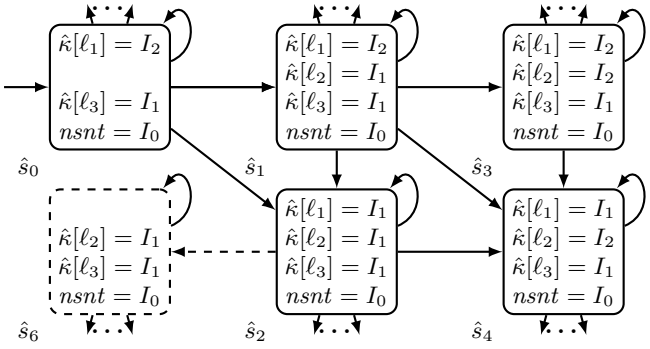


Fig. 4. A small part of the transition system obtained by counter abstraction of counter representations for all parameters.

is to replace integer counters by counters over a finite abstract domain. In our work, we use the same domain as in the PIA data abstraction in Section 2.1, e.g., for Algorithm 1, we use the domain of four intervals $I_0 = [0, 1[$ and $I_1 = [1, t+1[$ and $I_2 = [t+1, n-t[$ and $I_3 = [n-t; n]$. Figure 4 illustrates counter abstraction of counter representations for all parameter values. For instance, the abstract states $\hat{s}_0, \hat{s}_1, \hat{s}_3, \hat{s}_4$ represent the concrete states s_0, s_1, s_3, s_3', s_4 from Figure 3. The abstract state $\hat{s}_2$ represents states that do not appear for the parameter values in Figure 3, but occur, e.g., for $n = 4, t = 1, f = 1$.

For decrementing and incrementing counters, a counter abstraction introduces abstract operations. For instance, an increment of abstract value I_1 should overapproximate that a concrete value from the interval $[1, t+1[$ is incremented. Note that increment can result in the same interval I_1 or in the next interval I_2. Similarly, decrement either maintains or changes its abstract value. When decrement and increment maintain the counter values, the abstract transitions form self-loops, as one can see in Figure 4. Hence, abstract increment is not deterministic. In particular, applying an abstract increment to a counter does not have to change the counter value ever, which introduces spurious behavior, i.e., abstract paths that do not correspond to real paths.

Our PIA counter abstraction uses many ideas developed by Pnueli et al. [32]. Regarding the abstract domain, they focused on mutual exclusion and thus used the well-known "$(0, 1, \text{more})$" abstract domain, whereas we focus on FTDAs and use intervals with parametric boundaries.

In this way, we arrive at a system of a fixed number of counters that range over a finite domain, that is, a finite-state model checking problem. We have used this in [21] (cf. [16] for technical details) to check *safety and liveness* of classic fault-tolerant broadcasting algorithms under a number of fault models. As in [32], abstraction makes liveness verification more challenging as it requires to add *justice* constraints. Moreover, for liveness we had to deal with spurious counterexamples, cf. Section 2.4.

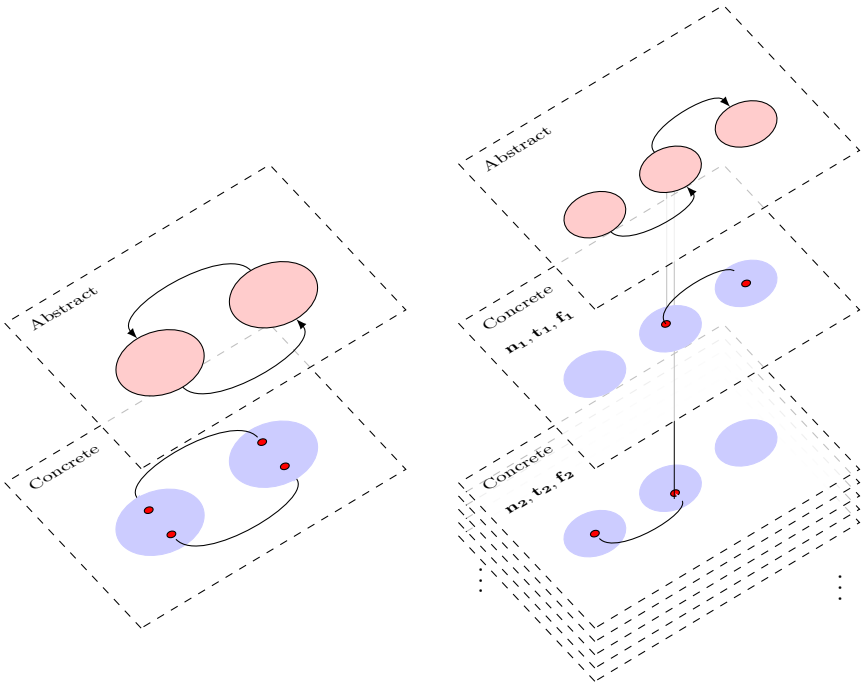


Fig. 5. Spurious loop due to to coarse abstraction in classic CEGAR [10] on the left, and a spurious path due to many concrete systems that are mapped to one abstract systems in parametrized model checking on the right.

2.4 Parametrized Abstraction Refinement

Our PIA abstraction maintains the relevant properties of threshold guards and counters, so that the classic CEGAR approach [10], which consists of refining the state space, is not suitable. However, the non-determinism due to abstract operations on counters leads to *spurious transitions* that lead to spurious counterexamples. Hence our abstraction refinement approach deals with removing transitions.

Our main problem stems from the non-determinism due to abstract counters. If a process moves from local state ℓ to ℓ' we have to decrease the counter $\kappa(\ell)$ and increase $\kappa(\ell')$. However, abstract decrease may lead to a smaller abstract value, while abstract increase maintains the counter value. Overall, processes may be lost. As we use global variables to record the number of message sent, we have the same effect there and messages "may be lost" due to abstraction. As messages may be required to make progress, this generates challenges for the verification of liveness properties.

Our current approach is to use an SMT solver to check whether abstract transitions correspond to concrete ones. If this is not the case, we explicitly remove these transitions from the transition relation of the counter abstraction.

See [16] for implementation details, where we also discuss how we refine unfair loops that occur in some case studies.

We would like to mention that abstraction refinement in parametrized model checking generates challenges different from classic CEGAR. As shown in Figure 5, abstract transitions that build a path in the abstract system may stem from different concrete systems for different parameter values. Currently, we deal with such counterexamples by user-provided invariant candidates that our tool checks to be invariants and which are then used for verification. To achieve more automation, one has to detect spurious paths instead of individual spurious transitions. However, this is challenging in the parameterized case, as infinitely many concrete systems are involved.

2.5 Threshold Automata

In Sections 2.1–2.4, we used control-flow automata (CFA) as an input to our model checking techniques (cf. Figure 1). A CFA is a formal presentation that is close to pseudo code and symbolically captures the transition relation of a single process as a formula over input, output (primed), and temporary variables. A path through the control-flow automaton (non-deterministically) computes a single transition in the transition relation of the algorithm. For instance, the leftmost path of the CFA shown in Figure 1 computes the local transition from the local state with the assignment $sv \mapsto \text{V0}$, $rcvd \mapsto 0$, and $nsnt \mapsto 0$ to the local state with the assignment $sv \mapsto \text{V0}$, $rcvd \mapsto f$, and $nsnt \mapsto 0$ (once the target state of the transition is computed, the primes are dropped). If we apply data abstraction (see Section 2.1) to the *local variables*, we obtain an abstract control-flow automaton. Likewise, a path through the abstract control-flow automaton computes a single transition in the abstract transition relation. There is, however, an important difference between the input CFA and the CFA that is created by data abstraction: the domain of the local variables, e.g., *rcvd*, in the abstract CFA is finite, and hence the local state space of each process is finite. This observation allows us to use another representation of the abstract transition relation, which we call a *threshold automaton* [23].

In a nutshell, a threshold automaton is a graph, whose nodes correspond to the abstract local states of a process, and the edges correspond to the local transitions. The edges are annotated with linear arithmetic constraints over the parameters and the shared variables, e.g., $nsnt \geq (n-t)-f$, as well as with increments of the shared variables, e.g., $nsnt' = nsnt+1$. Note the three important differences of a threshold automaton from a CFA:

1. The nodes of a threshold automaton correspond to the local states, whereas the nodes of a CFA correspond to the locations in the control flow of the code computing the next state of the algorithm;
2. An atomic step of the algorithm is represented by an edge of a threshold automaton, as opposite to a path of a CFA;
3. The edges of a threshold automaton are annotated only with shared variables and parameters, whereas the values of the local variables are implicit in the automata nodes.

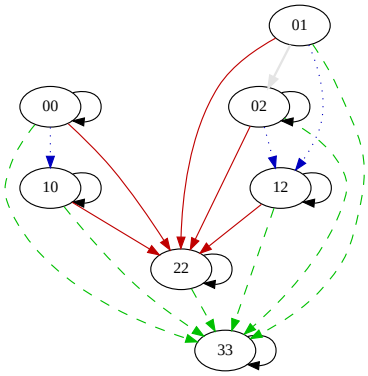


Fig. 6. A threshold automaton for the CFA shown in Figure 1. The nodes correspond to the local states of the processes, while the edges correspond to the guarded transitions. The edges are annotated with guards as follows: the bold gray edge is guarded with *true*; the dotted edges are guarded with $nsnt \geq 1 - f$; the solid edges are guarded with $nsnt \geq (t+1) - f$; the dashed edges are guarded with $nsnt \geq (n-t) - f$. Finally, $nsnt$ is incremented by the edges from the local states 00, 10, and 01 to the local states 12, 22, and 33, whereas all other edges do not change $nsnt$.

Figure 6 illustrates a threshold automaton that is constructed automatically from the CFA shown in Figure 1 by our tool. For instance, if a process is in local state 00 and $nsnt \geq (n-t) - f$, then the process may go to the local state 22. In doing so, it increases $nsnt$.

In our case studies, all increments of shared variables in threshold automata are outside of loops. This is a consequence of the class of FTDAs under consideration: each correct process sends a message of each type at most once, and thus increases each shared variable at most once. The partial order reduction techniques in Sections 2.6 and 2.7 exploit this property to guarantee completeness of bounded model checking.

2.6 Checking Reachability by Bounded Model Checking using Offline Partial Order Reduction and Acceleration

In [23], we apply SAT-based bounded model checking to verify reachability properties of the finite model obtained by counter abstraction of FTDAs (see Section 2.3). It is well-known that to make bounded model checking complete for reachability properties, one has to analyze executions of length up to the diameter of the transition system [3].

To this end, we first compute an upper bound on the diameter of the counter representation, that is, an upper bound on the minimal number of steps required to reach any configuration σ' from a configuration σ. From the bound on the counter representation we obtain a diameter bound on the counter abstraction. In the following we discuss why, surprisingly, the diameter is bounded.

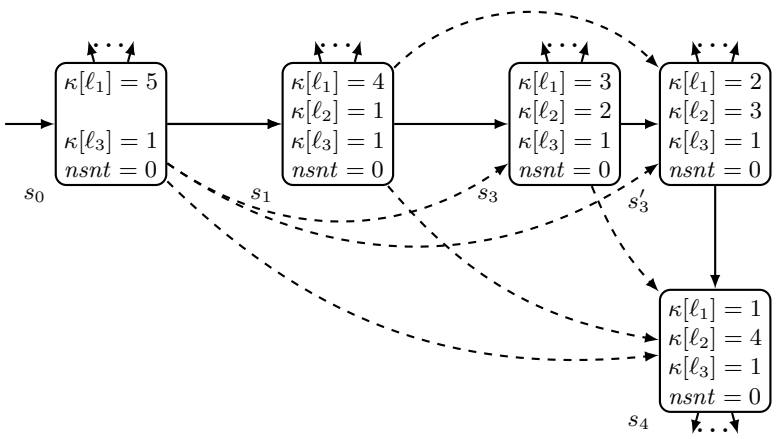


Fig. 7. A counter system in Figure 3 extended with accelerated transitions (dashed)

Assume σ' is reached from σ by steps of two processes where each process transitions from local state ℓ to local state ℓ'. In classic interleaving semantics, this run has length 2. However, we might also model this as a single update on the counters, that is we may decrease the counter $\kappa(\ell)$ and increase $\kappa(\ell')$ by two, respectively. This idea is illustrated in Figure 7. In general, we may move arbitrarily many process at once, and call such runs of counter systems *accelerated*. In this example, 2 would be the *acceleration factor*. In the context of parametrized model checking, the important property is that because we may move arbitrarily many process at a time, there is potential to bound the diameter *independently* of the value of the parameters!

Exploiting commutativity arguments not given in detail here, by swapping two neighboring transitions in a run, we obtain the same final state. To combine this with acceleration, one would like to swap transitions in such a way that many neighboring transitions can be accelerated. Importantly, one has to ensure that after swapping the guard of a transition still evaluates to true. Ensuring this has great influence on the actual bound and is the key technical argument from [23], where we also show that the resulting bounds are sufficiently small to check several case studies. Note that our method can be seen as a form of partial order reduction that is applied before model checking, i.e., an offline partial order reduction.

2.7 Bounded Model Checking using SMT

Our final method avoids counter abstraction and directly encodes runs of the counter representation in SMT. A global system state, which contains basically one counter per local state, can be represented as a vector of integer variables (one for each local state). As in SAT-based bounded model checking, one can

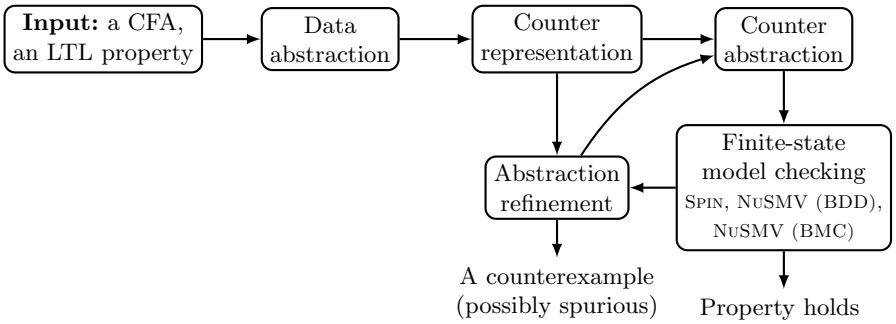


Fig. 8. Parameterized verification of FTDAs with data and counter abstractions [21,16,23]

then encode the transition relation, and the subsequent global state using a fresh vector of integer variables (or fresh integer variables for the counters that have actually been updated).

While the technique of Section 2.6 conceptually enumerates all runs of length up to the diameter, in [24] we only encode a small set of "schemas", and show that the (representative) runs generated from the schemas span the reachable state space. A schema is essentially a sequence of scheduling constraints containing guards. The schemas are obtained by an improvement of the partial order ideas that we used in [23] to bound the diameter. Thus, we obtain a more aggressive offline partial order reduction, and significantly better experimental results that are discussed in Section 4.

To illustrate schemas, consider the threshold automaton depicted in Figure 6. The automaton has three guards: $\varphi_1 \equiv nsnt \geq 1 - f$, $\varphi_2 \equiv nsnt \geq t + 1 - f$, and $\varphi_3 \equiv nsnt \geq n - t - f$. Consider the following transitions of the threshold automaton: the transition r_1 from 01 to 02; the transition r_2 from 02 to 22, the transition r_3 from 22 to 33. Then, a schema $\{\}r_1\{\varphi_1, \varphi_2\}r_1r_2r_3\{\varphi_1, \varphi_2, \varphi_3\}$ generates runs for various parameter values, where the transition r_1 is executed by several processes first and makes the guards φ_1 and φ_2 true; after that the transitions r_1, r_2, and r_3 are executed by several processes one after the other and make the guard φ_3 true.

The number of different threshold guards in the typical distributed algorithms in the literature varies from one to ten, which results in a reasonably large number of schemas that have to be checked, typically several thousand schemas [24]. Note that the schemas can be verified independently, and thus, in parallel.

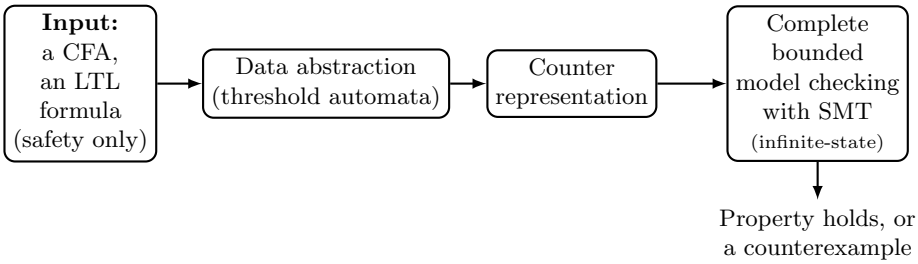


Fig. 9. Parameterized verification of FTDAs with data abstraction and SMT-based bounded model checking [24]

3 Implementation: Byzantine Model Checker

We have implemented the techniques described in Section 2 in our tool ByMC: Byzantine Model Checker[1]. Figures 8 and 9 illustrate two different workflows that combine our techniques within ByMC.

In the first workflow depicted in Figure 8, our tool computes data and counter abstractions and invokes a model checker to verify a finite-state abstract system. Depending on the choice of the model checker, ByMC can verify either safety properties, or both safety and liveness: the explicit-state model checker Spin [18] or the BDD-based symbolic algorithms in NuSMV/nuXmv [7] allow us to verify safety and liveness as described in [21]; the SAT-based bounded model checker implemented in NuSMV/nuXmv allows us to verify safety properties[2] as described in [23]. When a model checker reports a counterexample, ByMC checks whether the counterexample is spurious, and when it finds spurious behavior, ByMC refines the counter abstraction.

In the second workflow depicted in Figure 9, our tool computes only data abstraction (Section 2.1), constructs a threshold automaton (Section 2.5) and computes a complete set of schemas (Section 2.7) as described in [24]. Each schema is encoded as an SMT formula in linear integer arithmetic and checked with an SMT solver, e.g., Z3 [11] or MathSAT [9]. As this technique maintains precise process counters, it does not produce spurious counterexamples that are caused by counter abstraction in the first workflow. Thus, the refinement loop is not required in our experiments.

4 Evaluation and Case Studies

In Figures 10 and 11 we show how our techniques allowed us to check more and more involved distributed algorithms.

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

[2] Although NuSMV implements bounded model checking for LTL, our present results guarantee completeness only for safety properties.

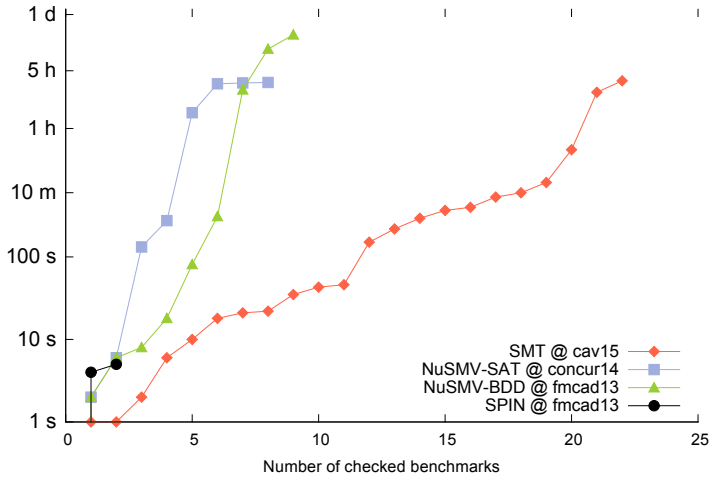


Fig. 10. Time to verify instances of fault-tolerant distributed algorithms

We are currently able to verify FTDAs that use threshold guards and work in asynchronous systems:

Broadcast. Reliable broadcast is a problem that can be solved in asynchronous systems, and we have verified the core of several such algorithms: Folklore reliable broadcast ("first forward to all then accept", e.g., given in [8]), Consistent Broadcast [35], Asynchronous Byzantine agreement [5]. Also the problem called "Condition-based consensus" can be solved in asynchronous systems and bears some similarities to broadcasting. We verified the condition-based consensus algorithm from [29]. After we published our verification results, a broadcasting algorithm very similar to [35] but with a different threshold was published in [19], and our tool easily checked its correctness.

FTDAs using failure detectors. The impossibility of solving non-blocking atomic commitment in asynchronous systems can be circumvented by using oracular mechanisms like failure detectors. They can be easily encoded in linear temporal logic. Thus, we verified such atomic commitment algorithms from [33,17].

Fast consensus algorithms. The idea of this class of algorithms is to have a quick (cheap) distributed preprocessing to a more expensive consensus algorithm: the algorithm terminates quickly in average runs, e.g., if there are no faults, if the system is not "too asynchronous", or if all processes have the same initial value. In case the preprocessing does not lead to a conclusive result, a "more-expensive" fall-back consensus algorithm is started with specific initial values. Our tool can check the correctness of this preprocessing of the algorithms BOSCO [34], C1CS [6], and CF1S [12].

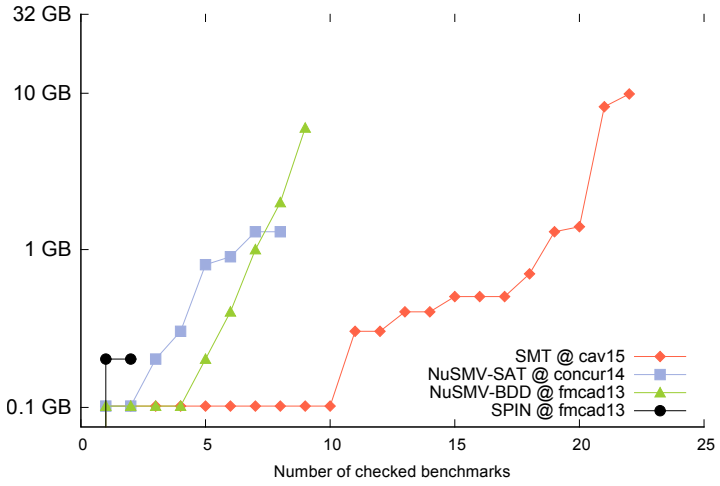


Fig. 11. Memory to verify instances of fault-tolerant distributed algorithms

Our techniques are currently limited to the class of asynchronous FTDAs that use only threshold guards. In particular, as consensus cannot be solved in asynchronous systems [15], we cannot completely verify algorithms for consensus, atomic broadcast, state machine replication, non-blocking atomic commitment, and similar hard problems. For that we need to restrict the interleavings and move from asynchronous systems to partially synchronous systems [14]. Only then, famous FTDAs like in [14] or Paxos [25] can be verified automatically in their entirety.

Our tool uses an extension of PROMELA as a front-end for CFA [22,16]. Their source code and the code of the threshold automata are freely available.[3]

5 Conclusions and Future Work

Automatic verification of fault-tolerant distributed algorithms is a challenging task. To the best of our knowledge, besides our own work, there are only few papers that deal with parameterized verification of FTDAs [1,13]. The main complications stem from multiple parameters, that are related by resilience conditions, as well as the fact that not only the number of processes, but also the code of each process is parameterized.

To make progress in automatic verification, our first steps have focused on domain-specific abstractions for a large class of fault-tolerant distributed algorithms with threshold guards. These guards are quite natural constructs in the distributed algorithms literature: for instance, majority voting on a value

[3] `https://github.com/konnov/fault-tolerant-benchmarks/`

is a natural technique to achieve agreement. The algorithms we address with our technique operate in the standard interleaving semantics (with fairness constraints). In terms of distributed algorithms literature, they are asynchronous. In the future, we will address also other computational models such as completely synchronous, partially synchronous, timed systems, and round-based systems.

Further, we want to develop more domain-specific techniques for increasingly larger classes of FTDAs. We are currently developing a tool[4] that implements these techniques and applies them to the popular TLA$^+$ specification language [27]. This will give us a framework and a toolset for verification of complex distributed algorithms such as Paxos [26].

Acknowledgments. We are grateful to Annu Gmeiner and Ulrich Schmid for their contributions to several papers [22,21,20,16] of our research agenda.

References

1. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. JSAT 8(1/2), 29–61 (2012)
2. Attiya, H., Welch, J.: Distributed Computing. Wiley, 2nd edn. (2004)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207 (1999)
4. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015)
5. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32(4), 824–840 (1985)
6. Brasileiro, F.V., Greve, F., Mostéfaoui, A., Raynal, M.: Consensus in one communication step. In: PaCT. LNCS, vol. 2127, pp. 42–50 (2001)
7. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. LNCS, vol. 8559, pp. 334–342 (2014)
8. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM 43(2), 225–267 (March 1996)
9. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS. LNCS, vol. 7795. Springer (2013)
10. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 1579, pp. 337–340 (2008)
12. Dobre, D., Suri, N.: One-step consensus with zero-degradation. In: DSN. pp. 137–146 (2006)
13. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: VMCAI. LNCS, vol. 8318, pp. 161–181 (2014)

[4] `http://forsyte.at/apalache/`

14. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J.ACM 35(2), 288–323 (1988)
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
16. Gmeiner, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In: Formal Methods for Executable Software Models. pp. 122–171. LNCS, Springer (2014)
17. Guerraoui, R.: Non-blocking atomic commit in asynchronous distributed systems with failure detectors. Distributed Computing 15(1), 17–25 (2002)
18. Holzmann, G.: The SPIN Model Checker. Addison-Wesley (2003)
19. Imbs, D., Raynal, M.: Simple and efficient reliable broadcast in the presence of Byzantine processes. CoRR abs/1510.06882 (2015), http://arxiv.org/abs/1510.06882
20. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Brief announcement: parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: PODC. pp. 119–121 (2013)
21. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD. pp. 201–209 (2013)
22. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards modeling and model checking fault-tolerant distributed algorithms. In: SPIN. LNCS, vol. 7976, pp. 209–226 (2013)
23. Konnov, I., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In: CONCUR. LNCS, vol. 8704, pp. 125–140 (2014)
24. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In: CAV (Part I). LNCS, vol. 9206, pp. 85–102 (2015)
25. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (May 1998)
26. Lamport, L.: Paxos made simple. ACM Sigact News 32(4), 18–25 (2001)
27. Lamport, L.: Specifying systems: The TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
28. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
29. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN. pp. 541–550 (2003)
30. Netflix: 5 lessons we have learned using AWS. (2010), http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html
31. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J.ACM 27(2), 228–234 (1980)
32. Pnueli, A., Xu, J., Zuck, L.: Liveness with (0,1,∞)- counter abstraction. In: CAV, LNCS, vol. 2404, pp. 93–111 (2002)
33. Raynal, M.: A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In: HASE. pp. 209–214 (1997)
34. Song, Y.J., van Renesse, R.: Bosco: One-step Byzantine asynchronous consensus. In: DISC. LNCS, vol. 5218, pp. 438–450 (2008)
35. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Dist. Comp. 2, 80–94 (1987)