HABILITATIONSSCHRIFT

Techniques and Tools for Automated Verification of Fault-tolerant and Parameterized Distributed Systems

ausgeführt zum Zwecke der Erlangung der Lehrbefugnis
für Informatik

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Igor Konnov, PhD

im Dezember 2018

HABILITATION THESIS

Techniques and Tools for Automated Verification of Fault-tolerant and Parameterized Distributed Systems

submitted to the

Faculty of Informatics of TU Wien

in December 2018

by

Igor Konnov, PhD

Abstract

Failures of distributed systems sometimes have drastic effects. Classical examples are networked embedded systems in flight control and automotive industry. Recent examples are cloud systems, which contain thousands of servers built of fault-prone commodity hardware. Reasoning about distributed systems of such scale is inherently hard, and human intuition is often defeated by the complexity of the task. A rigorous approach to verification of today's distributed systems has to address two questions: (i) How to verify a distributed system designed for a faulty environment, and (ii) How to verify a distributed system of real scale, e.g., with thousand components.

These questions are notoriously difficult. Hence, in state-of-the-art system development, the engineer's trust in fault-tolerant distributed systems is supported by two kinds of arguments (in addition to thorough system testing). First, to reason about safety and liveness of basic distributed algorithms, distributed algorithms designers write pencil & paper mathematical proofs that apply to all numbers of processes and faults. Second, to debug designs of distributed systems, engineers write high-level specifications and run model checkers on small instances, which comprise three or four processes and allow one fault to happen. The problem with this approach is that the actual distributed systems run hundreds of processes, and thus the classical model checking tools would miss the bugs that occur in the systems running these large numbers of processes. To guarantee reliability of such systems, engineers need verification tools that are both tailored to the mechanisms found in fault-tolerant distributed algorithms and scale to the realistic system sizes, or all system sizes.

The publications in this habilitation thesis present novel techniques for parameterized model checking of fault-tolerant distributed algorithms. We were the first to introduce techniques and tools for model checking of threshold-guarded algorithms such as reliable broadcast, non-blocking atomic commit, and one-step consensus. Prominently, our verification results have two features crucial for the algorithm designers: we have verified both safety and liveness, and our results hold for all numbers of processes and faults. Moreover, as we have recently shown, these results can be applied also to automated synthesis of fault-tolerant distributed algorithms.

This habilitation thesis consists of two journal articles and six peer-reviewed conference papers. Our new techniques contribute to model checking and parameterized verification by employing and extending such methods as *abstraction*, *reduction*, *acceleration*, *bounded model checking*, *satisfiability modulo theories*, *counterexample-guided abstraction refinement*, and *counterexample-guided inductive synthesis*.

Acknowledgments

First of all, I would like to thank my teachers and senior colleagues from whom I have learnt the most: Vladimir Zakharov, Helmut Veith, and Josef Widder. Vladimir Zakharov introduced me to the amazing world of logic and computer-aided verification and convinced me that there are plenty of things to discover. Helmut Veith taught me that research should also be fun. Helmut always wanted to explore the fields "where no one has gone before" — it was Helmut who initiated and supported the research that is presented in this thesis. Josef Widder taught me a lot about distributed algorithms, what they are and, more importantly, what they are not. I thank Josef for our endless scientific arguments that led us to this new perspective on verification of fault-tolerant distributed algorithms.

This work would not be possible without the vibrant atmosphere of the Forsyte group (TU Wien), which was nurtured by Helmut Veith, Georg Weissenbacher, and Laura Kovacs. Without doubt, the work presented in this cumulative habilitation owes to the contributions by my co-authors: Helmut Veith, Josef Widder, Marijana Lazić, Annu Gmeiner, Qiang Wang, Tomer Kotek, Simon Bliudze, Francesco Spegni, Roderick Bloem, Ulrich Schmid, and Joseph Sifakis. I am sure that this work also benefited from our lunch conversations with Andreas Holzer, Florian Zuleger, Moritz Sinn, Georg Weissenbacher, Swen Jacobs, Ayrat Khalimov, Ilina Stoilkovska, and Sasha Rubin. It goes without saying that these fruitful discussions were often accompanied by Wiener Schnitzel in Wiazhaus!

The presented research required generous financial support. This was provided by the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403) and the Vienna Science and Technology Fund (WWTF) through grants APALACHE (ICT15-103) and PROSEED (ICT12-059). In Austria I found that funding agencies can be efficient when they minimize the bureaucratic burden on the researchers. The computational results presented in Part III have been achieved using the Vienna Scientific Cluster (VSC-3).

I thank Anna Prianichnikova and Helmut Veith, who shared their wisdom about Austria and the world and were always there to help. To my deep regret, Helmut would not be able to read these lines.

Finally, I owe a debt of gratitude to my wife Marina for moving with me to Vienna — and later to Nancy, while she already had a good job, for learning her third foreign language (German) in less than a year, and for patiently going through all the other challenges. And thanks to our daughter Sofija for waiting for me, while I was travelling and when finishing this thesis.

Nancy, France, October 2018

Contents

Introduction **1**

Overview of the Results **9**

I Modeling of Fault-Tolerant Distributed Algorithms and Model Checking by Abstraction **39**

1 Towards modeling and model checking fault-tolerant distributed algorithms **41**

2 Accuracy of Message Counting Abstraction in Fault-Tolerant Distributed Algorithms **61**

3 Parameterized model checking of fault-tolerant distributed algorithms by abstraction **83**

II Parameterized and Bounded Model Checking of Threshold-Guarded Distributed Algorithms with SMT **93**

4 On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability **95**

5 Para^2: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms **111**

6 A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms **151**

III Parameterized Synthesis of Threshold-Guarded Distributed Algorithms **169**

7 Synthesis of Distributed Algorithms with Parameterized Threshold Guards **171**

IV Parameterized Extension of Behavior-Interaction-Priority Framework **193**

8 Parameterized Systems in BIP: Design and Model Checking **195**

V Supplementary Documents **213**

9 Curriculum Vitae **215**

10 List of Publications **221**

Introduction

Distributed systems are now everywhere: in airplanes, cars, phones, houses, and cloud services. All these systems consist of many interacting components, some of them can fail. One infamous source of failures are software bugs — logical errors introduced when writing code. Another source are the failures that are caused not by the programmer's oversight, but by the environment itself: power outages, disk failures, memory corruption, or network misconfiguration. Our personal computers and handheld devices have indeed become quite reliable — compare to the earliest computers like ENIAC that had a tube failing once in two days [Randall 2006] — and thus it is tempting to believe that building a distributed system out of these components should be easy. Unfortunately, this is not true. Failures occur every day in large distributed systems [Bailis and Kingsbury 2014] as well as in supercomputers [Geist 2016]. When a failure uncontrollably propagates throughout a distributed system, it may freeze the whole system. Designing *fault-tolerant* distributed systems has been a subject of basic research in distributed computing since the late 1970s [Lynch 1996, Attiya and Welch 2004]. New distributed algorithms are presented every year at PODC and DISC — premier conferences on distributed computing.

Fault-tolerance is achieved by replication. To mitigate at most f (Byzantine) faults, one builds a system $S(n, f)$ of $n > 3f$ replicas, out of which at least the $n - f$ correct replicas have to agree on the result of computation. Lamport, Shostak, and Pease introduced this as *the consensus problem* and presented first *consensus algorithms* for solving it [Pease et al. 1980]. The distributed computing literature comprises numerous consensus algorithms, to name a few: the famous Paxos by Lamport [1998], Practical Byzantine Fault Tolerance by Castro et al. [1999], and Raft by Ongaro and Ousterhout [2014]. Consensus algorithms lie at the heart of the replicated state machine approach [Schneider 1990, Lamport 1978, Lamport et al. 2010] and are run in distributed databases [Cor-

```
1 input: v_p
2 broadcast ⟨VOTE, v_p⟩ to all processors;
3 wait until n − t VOTE messages have been received;
4 if more than (n+3t)/2 VOTE messages contain the same value v
5 then DECIDE(v);
6 if more than (n−t)/2 VOTE messages contain the same value v,
7     and there is only one such value v
8 then v_p ← v;
9 Underlying−Consensus(v_p);
```

Figure 1: BOSCO: Byzantine one-step consensus by Song and van Renesse [2008]

bett et al. 2013], crypto currencies and distributed ledgers [Garay et al. 2015, Abraham et al. 2017], and cloud systems [Chandra et al. 2007].

An example of a fault-tolerant distributed algorithm. As a concrete example, consider the pseudo-code in Figure 1, which describes Byzantine one-step consensus (BOSCO) by Song and van Renesse [2008]. This algorithm works under the following assumptions:

(a) The distributed system comprises n processes, where n is a parameter. Every non-faulty process runs the code of the algorithm.

(b) The environment is asynchronous: There are no bounds on relative processor speeds as well as times of message delivery. (One assumes, however, that the non-faulty processes are scheduled infinitely often and that every message sent by a non-faulty process is eventually delivered to every non-faulty process.)

(c) At most $t < n/3$ processes may be subject to Byzantine faults, that is, behave in arbitrary way. For instance, they can stop or send arbitrary messages to the other processes.

(d) Each process p starts with an initial value v_p, e.g., a natural number.

(e) The processes broadcast their values to all other processes and count how many messages they have received.

(f) The processes have to eventually decide on exactly one value from the set $\{v_1, \ldots, v_n\}$.

As BOSCO runs in the asynchronous environment, there is no guarantee that every process would eventually decide; otherwise, it would have violated the famous impossibility result by Fischer et al. [1985]. Instead, BOSCO quickly solves consensus in the "good cases": (1) when $n > 7t$, or (2) when $n > 5t$ and there are no faults. In the other cases — when the processes cannot quickly decide — the processes fall back to a general consensus algorithm such as Paxos, which is called "`Underlying-Consensus`" in the pseudo-code. The underlying consensus algorithm should impose additional requirements on the environment, in order to guarantee termination.

Pseudocode and paper & pencil correctness proofs. It is standard in the research literature to write algorithms in pseudo code. By doing so, algorithm designers focus on algorithmic aspects and omit "book-keeping details" such as: receiving messages, expressing faults, or encoding communication constraints. Although pseudo code may appear simple, its specification in a formal language like TLA$^+$ [Lamport 2002] or its actual implementation are complex. For instance, the size of Raft code varies dramatically: the pseudo code is just 100 lines, the TLA$^+$ specification is 433 lines, and the implementation in `C++` is 31104 lines [Ongaro 2014].

It is common in the distributed computing literature to write mathematical proofs, in order to convince the peers that a new algorithm is correct. These proofs are usually sophisticated, as they require one to reason about concurrency, faults, and temporal behavior of the algorithm. Not surprisingly, the algorithms and their proofs may contain bugs. For instance, safety and liveness bugs were found in the earlier versions of Raft. While it is customary to catch these bugs by peer review, this process is error-prone and requires ingenuity. Thus, there is a need for automatic tools that would help algorithm designers in ensuring that their new algorithms are correct.

The techniques and tools of this habilitation thesis contribute to automatic reasoning about correctness of fault-tolerant distributed algorithms.

Formal specifications and interactive proofs. The need for formal specifications and rigorous correctness proofs of distributed algorithms was understood as early as in the 80ies, when Leslie Lamport and Nancy Lynch introduced their frameworks called Temporal Logic of Actions (TLA) [Lamport 1980; 2002] and

IO Automata [Lynch and Stark 1989, Lynch 1996], respectively. These frameworks were devised for manually-written proofs and later enhanced with proof assistants: PVS, Isabelle, and TLAPS. (They were also extended with model checkers, discussed below.) The strong point of TLA and IO automata is that, by design, they are not limited to a specific class of algorithms. But this freedom comes with a price tag: the user has to find invariants on the system state and prove that the distributed algorithm preserves the invariants. This is hard, as distributed algorithms — especially fault-tolerant algorithms — demonstrate high degree of concurrency and non-determinism.

Ironfleet [Hawblitzel et al. 2015] and Verdi [Wilcox et al. 2015] are two recent methodologies that encompass correctness proofs of fault-tolerant protocols and their implementations. In Ironfleet, state machines are specified in Dafny — an imperative, sequential language for functional verification with Z3 [Leino 2010, De Moura and Bjørner 2008]. The authors propose TLA-like refinement [Lamport 2002] to prove that a distributed protocol satisfies (i.e., refines) its high-level specification, and the implementation refines the protocol. In Verdi, the user specifies a distributed protocol and its implementation under assumptions of the perfect environment — no message losses, no process crashes. Verdi extends this system with fault-tolerance by applying transformation rules, whose correctness is proven with Coq [Bertot and Castéran 2004]. In both methods, the user invests substantial efforts in proof writing that is estimated in lines of code required to write: (a) the specification, (b) the implementation, and (c) the actual proofs. For the Ironfleet case studies, these figures are: (a) 1400, (b) 5114, and (c) 39253. For the Verdi case studies, the figures are: (a) 148, (b) 220, and (c) 2364.

PSync is a domain-specific language by Dragoi et al. [2016] that builds upon the Heard-Of model by Charron-Bost and Schiper [2009]. This language comes with a runtime environment and thus allows the algorithm designers to execute their algorithms. Moreover, the algorithms in PSync can be expressed in the logic called $\mathbb{CL}$ and verified with the semi-decision procedure that was introduced by Dragoi et al. [2014]. As usual, the user has to provide detailed enough inductive invariants and ranking functions, in order for the verification procedure to automatically prove the safety and liveness properties.

Recently, Padon et al. [2017] proposed a verification methodology based on effectively-propositional logic (EPR). In this methodology, the user has to specify the distributed algorithm and the invariant candidate in first-order un-

interpreted logic. In case this specification fits into the fragment of EPR, the verification tool verifies the invariance in a completely automatic way. Moreover, EPR enjoys a finite model property: If a formula in EPR is satisfiable, then there is a finite structure that serves as a model of the formula. Hence, the tool produces finite counterexamples, which reflect the nature of the analyzed systems. The crux of the approach is to fit the specification in EPR. This has to be done by the user, who has to extend the specification (and the invariant) with so-called derived relations. The purpose of these relations is to break cycles in the quantifier alternation graph, which prevent the corresponding formula from being in EPR. Padon et al. [2017] applied their methodology and the deductive verification tool IVy [McMillan 2016] to verify the invariants of several variations of Paxos.

Model checking. Automatic reasoning about non-deterministic and concurrent systems that belong to a specific application domain is a stronghold of model checking. Originally formulated in the 1980s by Clarke and Emerson [1981] and Queille and Sifakis [1982] as a graph traversal algorithm for proving that a finite-state machine M satisfies a temporal formula φ, model checking has been continuously evolving, e.g., see "Handbook of Model Checking" by Clarke et al. [2018]. For decades, research in model checking has been focused on combinatorial explosion that is caused — in different forms — by non-determinism, concurrency, and huge state spaces of state machines. Partial remedies to this problem are offered by the revolutionary concepts such as binary decision diagrams [Burch et al. 1990] and satisfiability solvers [Biere et al. 1999, Bradley 2012], partial order reduction [Godefroid 1990, Valmari 1991, Peled 1993], predicate abstraction [Graf and Saïdi 1997], and abstraction refinement using satisfiability-modulo-theory solvers [Clarke et al. 2003, Ball et al. 2011].

Despite critical role of fault-tolerant distributed algorithms, before our work, only a few techniques were developed for this domain. In a nutshell, an efficient model checker for fault-tolerant distributed algorithms has to address the following key problems:

1. General-purpose model checkers for distributed algorithms fall back to state enumeration, e.g., Spin [Holzmann 2003] and TLC [Yu et al. 1999], which suffers from state explosion.

2. Software model checking tools work best on sequential programs, which

have limited degree of non-determinism and no concurrency. That is, they cannot be directly applied to fault-tolerant distributed algorithms.

3. Many distributed algorithms, including consensus algorithms, are parameterized in the number of processes and faults and thus require *parameterized model checking*, e.g., proving $\forall n, f : \ n > 3f.\ S(n,f) \models \varphi$.

The problem of parameterized model checking is particularly hard. In fact, for many computational models, it is undecidable. In our recent survey, we reviewed state-of-the-art proofs and techniques for parameterized model checking [Bloem et al. 2015]. Interestingly, these techniques do not apply to fault-tolerant distributed algorithms, as they were developed for other domains: token-passing systems, (hardware-like) broadcast systems, centralized systems, cache coherency protocols, etc.

The techniques collected in this cumulative habilitation thesis rendered possible parameterized model checking of fault-tolerant distributed algorithms.

Related work. Classical model checkers such as Spin by Holzmann [2003] and TLC by Yu et al. [1999] enumerate reachable states of the state machine that specifies a distributed algorithm, or, more often, its abstraction. Such an abstraction is usually constructed manually by the verification expert with respect to the properties under analysis. These tools can verify distributed algorithms for a small number of processes, e.g., when run on a 8GB machine, Spin can verify Dolev-Klawe-Rodeh leader election in rings that contain up to eight processes [Attiya and Welch 2004][Ch. 3]. Similarly, Spin can verify Peterson's mutual exclusion for systems consisting of up to five processes [Lynch 1996][p. 284]. Owing to combinatorial explosion, state enumeration tools are able to check transition systems that have up to about a billion of states (depending on the problem and available memory).

Several consensus and agreement algorithms were verified with model checking for a fixed number of processes. Tsuchiya and Schiper [2011], Noguchi et al. [2012] used NuSMV and Spin to verify the LastVoting algorithm by Charron-Bost and Schiper [2009] and two consensus algorithms with failure detectors by Chandra and Toueg [1996], Mostéfaoui and Raynal [1999]. Delzanno et al. [2014] introduced a sophisticated encoding of Lamport's Synod algorithm [Lamport 2001] in Promela and checked its properties with Spin for systems of up

to six processes. They further proved manually that the properties hold in the parameterized case as well.

Besides the results presented in this thesis, there are only a handful of results on parameterized model checking of fault-tolerant distributed algorithms, that is, proving correctness for an unbounded number of processes n (and in some cases, for all possible faults $f < n/3$). Fisman et al. [2008] showed how to encode faults in the framework of regular model checking and manually reasoned about a broadcast algorithm for crash faults. Alberti et al. [2012] encoded synchronous broadcast algorithms in the theory of arrays and applied fix point computations to check these algorithms, though they could not handle arithmetic conditions such as $f < n/3$. Alberti et al. [2016] also considered SMT-based model checking of counter systems, following up our work [Konnov et al. 2015].

Kwiatkowska et al. [2001], Kwiatkowska and Norman [2002] verified several randomized distributed algorithms, including agreement and consensus algorithms. The almost-sure termination arguments were done with the probabilistic model checker PRISM [Hinton et al. 2006], for systems of 10–20 processes. They also used the Cadence SMV proof assistant to show safety of the non-randomized behavior of the distributed algorithms [Kwiatkowska et al. 2001], for a parameterized number of processes.

Marić et al. [2017] investigated consensus algorithms in a version of the "Heard-of" model by Charron-Bost and Schiper [2009], in which processes execute in rounds, but not necessarily in lock-step. They have shown that consensus algorithms in this model have a cut-off property: It is necessary and sufficient to verify these algorithms up to a precomputed number of processes and faults. (The cut-off property has been proven for the three properties of consensus, that is, agreement, validity, and termination.)

Recently, Aminof et al. [2018] introduced abstraction techniques for parameterized model checking of fault-tolerant distributed algorithms that work in the synchronous model of computation.

Overview of the Results

This cumulative habilitation thesis is organized in four parts:

Part I: Modeling of Fault-Tolerant Distributed Algorithms and Model Checking by Abstraction. We transfer the classical model checking techniques to fault-tolerant distributed algorithms. To this end, we model fault-tolerant distributed algorithms as systems of processes in PROMELA and check these models with SPIN for small numbers of processes and faults. We investigate soundness of this modeling under two different assumptions: (1) every message is eventually delivered, and (2) every message is delivered in a fixed time interval. Finally, we define *parametric interval abstraction* on message counters and process counters, which renders possible parameterized model checking of reliable broadcast.

This part includes three papers presented at: SPIN [John et al. 2013b], FMCAD [John et al. 2013c], and VMCAI [Konnov et al. 2017d]. The results were also reported as a brief announcement at PODC [John et al. 2013a] and a tutorial at SFM [Gmeiner et al. 2014], not included in this thesis.

Part II: Parameterized and Bounded Model Checking of Threshold-Guarded Distributed Algorithms with SMT. We introduce a radically new approach to parameterized verification of fault-tolerant distributed algorithms that count messages. Owing to problems with scalability of the techniques introduced in Part I, we introduce the new model of *threshold automata* and consider counter systems of threshold automata with a specific form of acceleration. For such counter systems we show that they have *bounded diameters*, which enables verification of the abstract systems from Part I with SAT-based bounded model checking.

We improve this result by showing that for reachability properties, it is necessary and sufficient to consider representative bounded executions, that is, the executions that follow precomputed *schemas*. Interestingly, one can encode

schemas in linear integer arithmetic and thus eliminate counter abstraction. In practice, this speeds up verification dramatically. Finally, we extend these results to safety and liveness properties that can be encoded in a fragment of linear temporal logic with F and G. Due to these results, we have verified safety and liveness of 10 threshold-guarded fault-tolerant algorithms.

Part II includes one conference paper presented at POPL [Konnov et al. 2017b] and two journal articles published in Formal Methods in Systems Design [Konnov et al. 2017a] and Journal of Information & Computation [Konnov et al. 2017c]. The journal articles extend the results that were presented at three conferences: CONCUR, CAV, and PSI [Konnov et al. 2014; 2015; 2016b]. For this reason, these three conference papers are not included in the thesis.

Part III: Parameterized Synthesis of Threshold-Guarded Distributed Algorithms. We consider the problem of automatically synthesizing guards in threshold automata. The approach introduced in Part II is crucial for synthesis: we have integrated Byzantine model checker with a synthesizer in the counterexample-guided inductive synthesis loop. Using this synthesis technique, we have automatically generated all possible threshold guards for reliable broadcast, hybrid broadcast, and Byzantine one-step consensus (BOSCO).

Part III includes the paper that was presented at the conference on Principles of Distributed Systems (OPODIS) [Lazic et al. 2017].

Part IV: Parameterized Extension of Behavior-Interaction-Priority Framework. We extend the framework of Behavior-Interaction-Priority (BIP) to the parameterized case. To this end, we introduce First-Order Interaction Logic (FOIL), which allows one to specify component interactions as first-order formulas. We show that FOIL can express rendezvous, synchronous broadcasts, token passing, etc. in parameterized systems. FOIL can be used for identifying parameterized verification techniques that can be applied to a parameterized BIP design.

Part IV includes the conference paper that was presented at CONCUR [Konnov et al. 2016a].

In the rest of this chapter, we put the results of Parts I–IV in perspective.

Overview of Part I

Chapter 1: Towards modeling and model checking fault-tolerant distributed algorithms [John et al. 2013b]

As was mentioned in the introduction, fault-tolerant distributed algorithms are usually published in pseudo-code, as exemplified with BOSCO in Figure 1 (page 2). Hence, the first step towards verification of such algorithms is their formalization. In Chapter 1, we propose to use PROMELA as a pragmatic specification language that is supported by the SPIN model checker. However, we noticed that the communication primitives offered by PROMELA are quite different from those that are used in the asynchronous fault-tolerant distributed algorithms, e.g., reliable broadcast by Srikanth and Toueg [1987b] and BOSCO. On one hand, PROMELA offers the user the following communication primitives: shared variables, synchronous message passing (rendezvous), and point-to-point asynchronous message passing (bounded FIFO channels). On the other hand, fault-tolerant algorithms use the *message counting* primitives: (1) send a message of a specific type to all other processes, and (2) count the number of distinct messages of a given type that were received from all the processes.

Main contributions. In Chapter 1, we introduce two approaches to formalization of message-counting: (a) *FIFO channels* and (b) *message counters*, that is, integer counters that accumulate the number of sent and received messages. We also present two approaches to modeling Byzantine processes: (a) *explicit faults*, that is, distinct processes send messages of predefined types in arbitrary order, and (b) *fault injection*, that is, the message counters are non-deterministically incremented, which models delivery of messages from the faulty processes.

Importantly, we do not consider rendezvous communication, as it can block a process on the receiving side, which can be exploited by an adversarial Byzantine process. In fact, even a crashing process would block a distributed system with rendezvous communication. This explains why rendezvous synchronization is typically not used in fault-tolerant distributed algorithms.

Following these modeling choices, we formalize several algorithms: folklore broadcast by Chandra and Toueg [1996], reliable broadcast by Srikanth and Toueg [1987b], Byzantine agreement by Bracha and Toueg [1985], and condition-based consensus by Mostéfaoui et al. [2003]. We checked small instances of these

algorithms with Spin. The experiments showed that Promela models with message counters and fault injection produce significantly smaller search spaces than the models with FIFO channels and explicit faults.

Modeling explained. It is also important that our modeling with message counters and fault injection supports parameterization in the number of processes and faults. This modeling is used in the parameterized verification technique of Chapter 3. As it is essential for the following chapters, we informally introduce it here using the example of BOSCO in Figure 1.

Consider the binary consensus, that is, every process is initialized with a value from the set $\{0, 1\}$, and eventually all the correct processes have to decide on the same value. Additionally, the processes should not decide on different values, and if a process decides on a value, this should be the initial value of one of the correct processes. In our modeling, the ith correct process maintains a *protocol counter* $pc(i)$ that ranges over a finite set of control locations, e.g., $init_0$ for being initialized with value 0, $sent_1$ for having sent value 1, $decide_0$ for having decided on value 0.

Further, the processes share an integer counter $nsnt_M$, one per message type M. So, in our example the processes have access to the counters $nsnt_{\langle \text{VOTE},0\rangle}$ and $nsnt_{\langle \text{VOTE},1\rangle}$, which accumulate the number of the messages $\langle \text{VOTE}, 0\rangle$ and $\langle \text{VOTE}, 1\rangle$ broadcast by the *correct processes*. Hence, the broadcast statement `broadcast <VOTE, v_p> to all processors` in line 2 of BOSCO is modeled as the increment of the shared variable, for instance:

$$\texttt{if} \ (pc(i) \ \texttt{==} \ init_0) \ \ nsnt_{\langle \text{VOTE},0\rangle}\texttt{++}$$

Apart from the protocol counter, every correct process has a local variable $nrcvd_M(i)$, one per a message type M. Such a variable maintains the number of messages of type M received from the distinct processes. At every algorithmic step of the process i, the next value of $nrcvd_M(i)$ is updated non-deterministically to model message delivery, for instance:

$$nrcvd_{\langle \text{VOTE},0\rangle}(i) \le nrcvd'_{\langle \text{VOTE},0\rangle}(i) \le nsnt_{\langle \text{VOTE},0\rangle} + f \qquad (1)$$

One way of implementing Equation 1 in Promela is by non-deterministically incrementing the received variable for a message type x:

```
next_rcvd_x = rcvd_x;
if
  :: next_rcvd_x < rcvd_x + f -> next_rcvd_x++;
  :: skip; /* do nothing */
fi;
```

Alternatively, in the later versions of our modeling, we are using the following symbolic constraint, which we introduced in our parametric extension of Promela, e.g., see [Konnov and Widder 2018]:

```
assume(next_rcvd_x >= rcvd_x);
assume(next_rcvd_x <= nsnt_x + f);
```

Note that a correct process may receive up to f messages from the faulty processes, although it does not have to. Since the distributed algorithm is designed for the model of reliable communication, we impose a fairness constraint for every message type saying that eventually the number of messages received by a correct process is at least as large as the number of messages sent by the correct processes, for instance:

$$\mathsf{F}\,\mathsf{G}\ nrcvd_{\langle \text{Vote},0\rangle}(i) \geq nsnt_{\langle \text{Vote},0\rangle}$$

Finally, a comparison such as "$n-t$ `VOTE messages are received`" in line 3 is modeled as an arithmetic comparison over the local variables and the parameters:

$$nrcvd_{\langle \text{Vote},0\rangle}(i) + nrcvd_{\langle \text{Vote},1\rangle}(i) \geq n - t$$

Chapter 2: Accuracy of Message Counting Abstraction in Fault-Tolerant Distributed Algorithms [Konnov et al. 2017d]

In Chapter 1, we have introduced modeling with message counters. Although it seems intuitively clear that this modeling is equivalent to the modeling that stores sent and received messages in message buffers or sets, we did not give a proof of soundness there. In Chapter 2, we give a formal argument for this equivalence: *There is a bisimulation between asynchronous message-passing models that use message counters and models that use message sets.* As a result, the models using message counters and the models using message sets satisfy the same $\mathsf{CTL}^\star$-formulas.

Main contributions. The main contributions of Chapter 2 are concerned with message-passing with delays. In this model, the time of message delivery lies in the interval $[\delta^-, \delta^+]$ for fixed constants $\delta^-, \delta^+ \in \mathbb{R}$. This model is typical for clock synchronization algorithms and is used, for example, in the optimal clock synchronization by Srikanth and Toueg [1987a] (they assume $\delta^- = 0$). To this end, we introduced timed automata of two kinds: (MPTA) encoding message-passing with sets and (MCTA) encoding message counters.

It would be natural to expect that the systems of MPTA should be equivalent to the systems of MCTA, as in the asynchronous case considered above. However, timed automata use a more refined notion of equivalence that takes duration of transitions into account (clocks are uniformly advanced in time automata). *Timed bisimulation* [Čerāns 1993] and *time-abstracting* bisimulation [Tripakis and Yovine 2001] offer such equivalences in the framework of time automata. Interestingly, in Chapter 2 we show that *neither timed bisimulation, nor time-abstracting bisimulation are preserved when one replaces message sets with message counters.* Roughly speaking, message counters abstract away the identities of the processes that have sent messages, and thus introduce new orders, in which messages may be delivered.

We prove in Chapter 2 that there is a way to construct *timed simulations* between systems of MPTA and MCTA, *in both directions.* Thus, the systems of message-counting timed automata satisfy the same ATCTL-formulas as the systems of message-passing timed automata. (Formulas in ATCTL quantify over *all paths*, as opposite to formulas in ETCTL that quantify over *some paths.*) In fact, model checkers for timed automata such as UPPAAL [Behrmann et al. 2006] support a subset of formulas from ATCTL. Hence, message counting preserves sufficiently many properties for model checking.

Chapter 3: Parameterized model checking of fault-tolerant distributed algorithms by abstraction [John et al. 2013c]

In this chapter we are using the modeling introduced in Chapter 1 and consider the following *parameterized model checking problem*:

$$\forall n, t, f.\ RC(n,t,f) \Rightarrow S(n,t,f) \models \varphi(n,t,f) \qquad (2)$$

In this problem, $S(n,t,f)$ corresponds to a distributed system of n *identical* processes, up to f of which are faulty, and t is an upper bound on f. The

distinction between t and f is important, as the process code can refer to the upper bound t, but cannot refer to the actual number of faulty processes f, which is unknown to the correct processes. Some combinations of parameters are not interesting in practice:

- The numbers of faults $f = -1$ and $f = 2n$ are not realistic.
- There are too many faults for the distributed problem to be solvable, e.g., when the upper bound on the number of Byzantine faults is $t \geq \frac{n}{3}$, agreement cannot be solved [Pease et al. 1980].

Thus, the scope of the parameterized model checking problem in Equation (2) is restricted with a *resilience condition* $RC(n, t, f)$.

Finally, $\varphi(n, t, f)$ is a formula in $\mathsf{LTL} \setminus \mathsf{X}$: linear temporal logic without the next-time operator. Importantly, the atomic predicates in φ can refer to the shared variables such as $nsnt$, the parameters n, t, f, and formulas indexed over protocol locations, e.g., $[\exists i.\ pc(i) = decide_0]$ or $[\forall i.\ pc(i) = decide_1]$.

The problem in Equation (2) poses two challenges:

1. As in the classical parameterized model checking, (cf. Bloem et al. [2015]), the specification φ must be checked for all possible instances of S.
2. In contrast to the classical parameterized model checking, the code of a single process uses the parameters n, t, and f. Although the transition system of a single process is finite for every choice of parameter values, it varies with the parameters.

We address both of these challenges by applying a new form of parametric interval abstraction in two steps: (1) abstract message counters $nrcvd_M(i)$ and $nsnt_M$ in the process code to obtain uniform and finite-state process code (*data abstraction*), and (2) abstract the integer process counters that keep the number of processes in each local state (*counter abstraction*).

To this end, we introduce the abstract domain of parametric intervals. For instance, the domain $\mathcal{D}$ introduced for BOSCO contains the following intervals[1]:

$$
\begin{array}{lll}
[0, 1), & [1, t+1), & \left[t+1, \left\lceil \frac{n-t+1}{2} \right\rceil\right), \\
\left[\left\lceil \frac{n-t+1}{2} \right\rceil, \left\lceil \frac{n+3t+1}{2} \right\rceil\right), & \left[\left\lceil \frac{n+3t+1}{2} \right\rceil, n-t\right), & [n-t, \infty).
\end{array}
$$

[1] BOSCO requires us to consider several domains that differ in the order of the thresholds. For simplicity, we give only the domain $\mathcal{D}$.

The interval bounds are parametric and are extracted from the threshold guards that are present in a distributed algorithm. (The interval $[0, 1)$ being an exception, which allows us to distinguish absence of processes in a local state from presence of processes in a local state.) These abstract domains generalize the domain $\{0, 1, \infty\}$, which was introduced by Pnueli et al. [2002].

Data abstraction. Using the domain $\mathcal{D}$, we define an abstraction function that maps concrete values to the intervals from $\mathcal{D}$ and translate the expressions over the variables $nrcvd_M(i)$ and $nsnt_M$ to expressions over the abstract variables $\widehat{nrcvd}_M(i)$ and $\widehat{nsnt}_M$ and over the abstract values $I_{[0,1)}, \dots, I_{[n-t,\infty)}$. This translation is done with the help of an SMT solver. For instance, the comparison $nrcvd_{\langle \text{VOTE},0\rangle}(i) < t + 1$ becomes $\widehat{nrcvd}_{\langle \text{VOTE},0\rangle}(i) = I_{[0,1)} \vee \widehat{nrcvd}_{\langle \text{VOTE},0\rangle}(i) = I_{[1,t+1)}$. By doing so, we obtain process code that (1) is independent of the parameters, and (2) produces finitely-many local states.

Counter abstraction. As data abstraction produces processes with finitely-many states, we can enumerate all the local states in a set $\mathcal{L} = \{\ell_1, \dots, \ell_m\}$ and switch to the counter representation. That is, instead of representing a system state of n processes as a tuple $(\ell_{s(1)}, \dots, \ell_{s(n)}, g)$, where g keeps the values of the shared variables, we represent the system state with the shared state g and m integer counters $\kappa_1, \dots, \kappa_m$ that sum up to $n - f$.[2] This counter representation captures the parameterized system: By choosing the parameter values and counters values in an initial state, we define a subspace of the states that corresponds to the chosen parameters. Importantly, this is just a change of representation, not an overapproximation, as the system processes are anonymous. (The processes may have identifiers, but they are not used in the code.)

Note that the counter representation has infinitely many states, though when fixing the parameters, it still produces finitely many states. To map the counter representation to a finite state system, we apply the parametric interval abstraction over the abstract domain $\mathcal{D}$ to the counters $\kappa_1, \dots, \kappa_{|\mathcal{L}|}$ and construct the counter system over abstract counters $\widehat{\kappa}_1, \dots, \widehat{\kappa}_{|\mathcal{L}|}$. This gives us an abstract finite state system that can be checked with an of-the-shelf model checker such as Spin or NuSMV.

[2] In case of Byzantine faults, we explicitly model $n - f$ correct processes and "inject" messages from the faulty processes. For the crash faults, we would have n processes and a distinguished "crashed" local state.

Counterexamples to liveness. Not surprisingly, our abstraction may produce spurious counterexamples, that is, counterexamples in the abstract system that do not have corresponding behavior in the concrete instances of the distributed algorithm. In our experiments, these counterexamples were produced when checking liveness properties. Essentially, there were two sources of spurious behavior: (1) too coarse counting caused by counter abstraction, and (2) unfair loops that were produced by too coarse (abstract) fairness constraints. Interestingly, regarding (2), similar effect of counter abstraction on fairness was investigated by Pnueli et al. [2002] for their $\{0, 1, \infty\}$-counter abstraction. To cope with the spurious counterexamples, we have introduced an abstraction refinement loop (following the CEGAR approach by Clarke et al. [2003]). The refinement loop works in practice, though it is not guaranteed to terminate. Indeed, the space of counter representation is infinite.

Implementation and experiments. We implemented this abstraction in the early version of *Byzantine model checker* (ByMC) and, for the first time, automatically verified safety and liveness of reliable broadcast by Srikanth and Toueg [1987b] in presence of various kinds of faults: Byzantine, crash, symmetric, and omission faults.

Overview of Part II

In part I, we followed the classical approach to model checking: we specified the algorithms in Promela and introduced abstractions to reduce the parameterized model checking problem to finite-state model checking. This approach worked for relatively simple algorithms such as folklore broadcast and reliable broadcast. However, it did not scale to more advanced algorithms such as condition-based consensus and BOSCO; the latter is shown in Figure 1. In this part, we introduce efficient techniques that are tailored to threshold-guarded fault-tolerant distributed algorithms. These techniques build upon three key observations:

1. *Threshold automata.* This model expresses threshold-guarded algorithms in terms of automata over finitely-many local states and shared message counters. It has no explicit local message counters. Hence, the distributed systems that run threshold-guarded algorithms can be directly expressed as counter systems. We introduced this model in [Konnov et al. 2014].

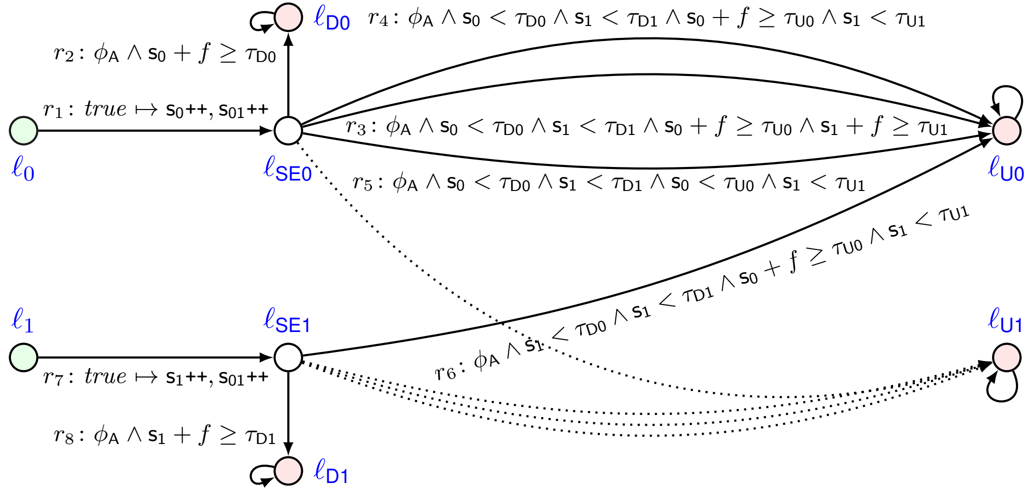


Figure 2: A threshold automaton for one-step Byzantine consensus (BOSCO). Labels of dashed edges are omitted; they can be obtained from the respective solid edges by swapping 0 and 1. The expressions τ_{D0}, τ_{D1}, τ_{U0}, and τ_{U1} are threshold expressions defined as follows: $\tau_{D0} = \tau_{D1} = \lceil \frac{n+3t+1}{2} \rceil$ and $\tau_{U0} = \tau_{U1} = \lceil \frac{n-t+1}{2} \rceil$

2. *Acceleration.* We introduce a domain-specific form of acceleration, which allows several processes to perform the same global step together. For instance, several processes may broadcast a message of the same type in a single step. This acceleration allows us to compress long executions into shorter ones.

3. *Complete bounded model checking.* We analyze bounded executions of counter systems produced by threshold automata. Interestingly, by employing reduction and acceleration arguments, we show that it is necessary and sufficient to analyze executions up to a certain pre-computable bound. Similarly for liveness, we analyze lassos of bounded length.

Threshold automata

As Chapters 4–6 use the model of threshold automata, we informally introduce threshold automata in this section. The formal definitions can be found in Chapter 4. Figure 2 shows a threshold automaton that encodes the BOSCO algorithm, which was introduced in Figure 1 (page 2).

A threshold automaton models a single correct process. Technically, a threshold automaton is a directed multigraph with finitely many nodes and edges. Its nodes are called *locations* and the edges are called *rules*. The locations model local states of a distributed algorithm, whereas the rules model transitions that are performed by one or several processes that follow the distributed algorithm. For instance, in BOSCO, the locations encode the local states of processes as follows:

- The locations ℓ_0 and ℓ_1 model the initial states of the correct processes that have 0 and 1 on their input respectively.
- The locations ℓ_{SE0} and ℓ_{SE1} model the intermediate states of the correct processes that have sent 0 and 1 respectively, but have not decided yet.
- The locations ℓ_{D0} and ℓ_{D1} model the states of the correct processes that have decided on 0 and 1 respectively.
- The locations ℓ_{U0} and ℓ_{U1} model the states of the correct processes that have called the underlying consensus with values 0 and 1 respectively.

A configuration of a distributed system is modeled as a vector of natural numbers that contains the values of: (1) system parameters such as n, t, and f; (2) message counters such as s_0, s_1, and s_{01}; (3) process counters such as $\kappa[\ell_0]$, $\kappa[\ell_{\mathsf{D0}}]$, etc. Similar to modeling in Part I, the message counters accumulate the number of messages that were sent by the correct processes. In our example, the (global) message counters s_0 and s_1 accumulate the number of $\langle \mathtt{VOTE}, 0\rangle$ and $\langle \mathtt{VOTE}, 1\rangle$ messages that were sent by the correct processes respectively, whereas the message counter s_{01} accumulates the sum of s_0 and s_1.

The automaton rules are labeled with threshold guards and increments. A threshold guard compares a message counter, such as s_0, against a linear combination of parameters, such as $n-t-f$. In general, every rule can be labelled with a conjunction of threshold guards. An increment action such as s_1++ prescribes the automaton to increment the global message counter by one.

In our example, the threshold automaton has the following guards that encode the conditions of the pseudo-code in Figure 1:

- The guard ϕ_A is defined as $\mathsf{s}_{01} + f \geq n - t$. This guard corresponds to the condition in line 3 of the pseudo-code: "$n - t$ VOTE *messages have been received*".

- The guards $\mathsf{s}_0 + f \geq \tau_{\mathsf{D0}}$ and $\mathsf{s}_1 + f \geq \tau_{\mathsf{D1}}$, where both thresholds τ_{D0} and τ_{D1} are defined as $\frac{n+3t+1}{2}$. These guards correspond to the condition in line 4 of the pseudo-code: "*more than* $\frac{n+3t}{2}$ `VOTE` messages contain the same value v".

- The guards $\mathsf{s}_0 + f \geq \tau_{\mathsf{U0}}$ and $\mathsf{s}_1 + f \geq \tau_{\mathsf{U1}}$, where both thresholds τ_{U0} and τ_{U1} are defined as $\frac{n-t+1}{2}$. These guards correspond to the condition in line 7 of the pseudo-code: "*more than* $\frac{n-t}{2}$ `VOTE` messages contain the same value v".

- The guards $\mathsf{s}_0 < \tau_{\mathsf{D0}}$, $\mathsf{s}_1 < \tau_{\mathsf{D1}}$, $\mathsf{s}_0 < \tau_{\mathsf{U0}}$, and $\mathsf{s}_1 < \tau_{\mathsf{U1}}$. These guards are used to encode the "else" branches in the pseudo-code.

Two comments are in order. First, as is usual in the distributed computing literature, we write thresholds in rational arithmetic, e.g., $\frac{n+3t+1}{2}$. This is done purely for the presentation purposes — these guards can be encoded in linear integer arithmetic. Second, the guards like $\mathsf{s}_0 + f \geq \tau_{\mathsf{D0}}$ contain the summand f, while the guards like $\mathsf{s}_0 < \tau_{\mathsf{D0}}$ do not. This concerns with a very subtle issue of modeling Byzantine faults. When a process is testing, whether it has received at least x messages, f Byzantine processes can change the outcome of the guard by sending up to f messages. However, when a process is testing, whether it has not received x messages, the Byzantine processes can change the outcome of the guard by not sending messages at all.

Importantly, we impose several constraints on the structure of threshold automata. Most prominently, a threshold automaton is not allowed to increment a shared variable in a cycle. Indeed, that would correspond to a correct process sending a message multiple times, which is not useful in the computational model of reliable communication.

If a rule is enabled in a global configuration, that is, its guard evaluates to true and the counter of the source location is positive, then the rule can move the system into a new configuration. There are two kinds of transitions: (1) a non-accelerated transition that corresponds to the rule fired by one process, and (2) an accelerated transition that corresponds to the rule fired by $m > 1$ processes.

In the non-accelerated case, the rule decrements the counter of the source location, increments the counter of the target location and increments the shared variables, as prescribed by the rule's actions. (Obviously, when a rule forms a

self loop, the location counters should not change.) For instance, the rule r_2 decrements $\kappa[\ell_{\mathsf{SE0}}]$ and increments $\kappa[\ell_{\mathsf{D0}}]$, provided that the threshold guards ϕ_A and $\mathsf{s}_0 + f \geq \phi_{\mathsf{D0}}$ hold true, and $\kappa[\ell_{\mathsf{SE0}}] > 0$.

In the accelerated case, the rule decrements the counter of the source location by m, increases the counter of the target location by m and, if shared variables are incremented by the rule, increases the respective shared variables by m. This can be done if: (a) the rule guard holds true for all acceleration factors between 1 and $m-1$, and (b) the value of the counter in the source location is at least m (obviously, this condition should be ignored for self loops). Number m is called *acceleration factor*. It can vary from transition to transition and can be arbitrarily large. Clearly, a non-accelerated transition can be seen as a special case of an accelerated transition. Hence, for a rule r and an acceleration factor $m > 0$, we write r^m to denote an accelerated transition performed by m processes. Finally, with r^0 we denote the vacuous transition doing nothing.

We call a sequence of transitions a *schedule*. Given a configuration σ and a schedule τ, one can define the notions of τ being applicable to σ, and, if τ is applicable to σ, the resulting configuration $\tau(\sigma)$. Schedules represent executions of a distributed system.

Our notion of acceleration is quite natural in the context of threshold-guarded distributed algorithms. More importantly, it allows us to design efficient model checking techniques, as explained below.

Chapter 4: On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability

We focus on the problem of parameterized reachability for counter systems of threshold automata. More precisely, given a threshold automaton TA and two predicates *Init* and *Bad* on the configurations of the respective counter system, we ask the following question:

Are there a configuration σ, for some values of the parameters, and a schedule τ that together have the following properties:

1. *Configuration σ is an initial configuration: $Init(\sigma)$ holds true,*
2. *there is a path from σ that follows the schedule τ, and*
3. *$Bad(\sigma')$ holds true for the last configuration $\sigma' = \tau(\sigma)$.*

Note that the configurations include the values of the parameters, and thus, the above question is parameterized in the number of processes and faults. An instance of this question is verification of the agreement property in BOSCO: Is there an execution that leads to a global state, in which one process decides on value 0, whereas another process decides on value 1? If this question is answered positively, then the agreement property does not hold, and σ and τ provide us with a counterexample. Otherwise, the property holds for all parameter values. This verification question can be encoded by choosing the following predicates:

$$
\begin{aligned}
\mathit{Init} &\equiv n > 3t \wedge t \geq f \geq 0 \\
\mathit{Bad} &\equiv \kappa[\ell_{\mathsf{D0}}] \neq 0 \wedge \kappa[\ell_{\mathsf{D1}}] \neq 0
\end{aligned}
$$

We observe that every schedule of a counter system of threshold automata has only a bounded number $\mathcal{C}$ of transitions that can "unlock" (or "lock") other rules, that is, by increasing the value of a shared variable a rule may enable (or disable) another rule of a threshold automaton. The number $\mathcal{C}$ can be computed from the structure of a threshold automaton.

Main contributions. The main contribution of Chapter 4 is in showing that the counter systems of threshold automata have bounded diameters (Theorem 8). Moreover, we give an upper bound on the diameter *diam* of a counter system of a threshold automaton TA: It is $(\mathcal{C}+1)\cdot|\mathcal{R}|+\mathcal{C}$, where $\mathcal{R}$ is the number of automata rules. By noticing that $\mathcal{C} \leq |\mathcal{R}|$, we further conclude that $\mathit{diam} \leq |\mathcal{R}|^2$.

In the proof of Theorem 8 [Konnov et al. 2017c][p. 17], we show that every schedule can be split into $\mathcal{C}+1$ segments that are joined by "milestone" transitions. By applying reordering arguments, we show that the transitions inside segments can be sorted with respect to the topological order of the rules in the threshold automaton. This kind of reasoning is inspired by the reduction arguments by Lipton [1975], but it is applied in the domain of threshold automata. As the threshold automaton may contain cycles, we introduce a preprocessing step that eliminates fruitless repetitions of the cycles.

Implementation and experiments. As a practical application of this theoretical result, we computed the diameters of the counter abstraction, which was introduced in Part I, and ran NuSMV to verify properties of more complex threshold-guarded distributed algorithms. This approach did not scale signifi-

cantly better than the approach from Chapter 3. However, it laid the ground for the efficient reduction-based techniques of Chapters 5 and 6.

Chapter 5: Para^2: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms

The result of Chapter 4 tells us that it is necessary and sufficient to test executions whose length is bounded by a precomputed number. While we used the result to check counter abstractions up to the diameter bound, the proof of Chapter 4 applies to the counter systems, that is, systems where counters are non-negative integers, not the abstract values. In this chapter, we encode the parameterized reachability problem directly as a bounded model checking problem in linear integer arithmetic. By doing so, we eliminate counter abstraction and thus eliminate the refinement loop that was necessary in Chapter 3. In comparison to the abstraction technique of Chapter 3, the technique presented in this chapter is not only theoretically sound, but is also complete: (1) the bounded model checking problem is encoded in a decidable theory, that is, linear integer arithmetic, and (2) the diameter bound gives us a completeness threshold for reachability properties (cf. [Clarke et al. 2004]).

However, there remains one obstacle. A straightforward encoding of bounded model checking in linear integer arithmetic requires us to encode a non-deterministic choice of the rule applied at every step of an execution. In this chapter, we introduce more refined reduction arguments that allow us to avoid this non-deterministic choice. We show that in order to explore all the bounded executions, it is necessary and sufficient to explore only the executions that can be generated by a *finite set* of so-called *schemas*.

In a nutshell, a schema is a sequence of rules interleaved with a sequence of sets of threshold guards, which should get unlocked or locked in the course of an execution. In our example of BOSCO in Figure 2, the following schema captures some of the executions that may lead two processes to the locations ℓ_{D0} and ℓ_{D1} and thus violate the agreement property:

$$\{\}\ r_1\ \{\mathsf{s}_0 + f \geq \tau_{\mathsf{D0}}\}\ r_7\ \{\mathsf{s}_0 + f \geq \tau_{\mathsf{D0}},\ \mathsf{s}_1 + f \geq \tau_{\mathsf{D1}}\}$$
$$r_1, r_7\ \{\phi_{\mathsf{A}},\ \mathsf{s}_0 + f \geq \tau_{\mathsf{D0}},\ \mathsf{s}_1 + f \geq \tau_{\mathsf{D1}}\}\ r_2, r_8 \quad \text{(S1)}$$

Schema (S1) starts with all guards disabled, executes an accelerated transition according to rule r_1, which enables the guard $\mathsf{s_0} + f \geq \tau_{\mathsf{D0}}$, executes an accelerated transition following the rule r_7 and so on. It is easy to see that schemas can be easily encoded in linear integer arithmetic. One introduces the counters for all the intermediate configurations and the acceleration factors and encodes the constraints that are imposed by the threshold guards and the updates of the counters. By conjoining these constraints with the violation of agreement, that is, $\kappa[\ell_{\mathsf{D0}}] \neq 0 \wedge \kappa[\ell_{\mathsf{D1}}] \neq 0$, we produce a query that states existence of an execution that follows schema (S1) and violates agreement. Satisfiability of this query can be checked with an SMT solver. If the query is satisfiable, a satisfying assignment constitutes a counterexample to the property.

Main contributions. We show that for every counter system of threshold automata, there is a finite set of schemas $\mathcal{S}$ with the following property (Theorem 4.2): Every configuration that can be reached by an execution of the counter system can be also reached by an execution that is generated by a schema from the set $\mathcal{S}$.

Moreover, we show how to construct such a set of schemas. Although this construction follows the main reduction principles formulated in Chapter 4, it introduces reduction of multiple executions, in contrast to a single execution. One technical difficulty of this construction is in finding schemas that would capture cycles that originate from different locations and end up in different locations. We have solved this by embedding spanning trees into schemas. As a result, the parameterized reachability in counter systems of threshold automata is reduced to a finite set of SMT queries.

Implementation and experiments. We have implemented this technique in our tool ByMC and verified safety of 10 threshold-guarded distributed algorithms, including the complex ones: BOSCO by Song and van Renesse [2008], condition-based consensus by Mostéfaoui et al. [2003], consensus in one communication step by Brasileiro et al. [2001]. This technique proved to be a dramatic improvement over the techniques from Chapters 3 and 4. This is one of the state-of-the-art techniques that are implemented in the latest version of ByMC. These experiments have successfully passed the artifact evaluation at CAV'15 [Konnov et al. 2015].

Chapter 6: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms

The results of Chapter 5 made it possible to verify safety of threshold-guarded distributed algorithms. (More precisely, the results allow us to verify those safety properties whose violation can be expressed with reachability.) However, it is a folklore knowledge in distributed computing that a safe distributed algorithm is quite easy to design: An algorithm that is doing nothing is always safe. Hence, to convince the algorithm designer that her algorithm is solving a distributed problem, one has to verify both safety and liveness of a distributed algorithm.

In this chapter, we extend the schema-based approach to liveness properties. We faced the following challenges:

1. *Finding a good logic.* In our early work [John et al. 2012], we considered $\mathsf{LTL} \setminus \mathsf{X}$ over the atomic propositions that quantify over protocol counter, e.g., $[\forall i.\ pc(i) = \mathsf{D0}]$ and $[\exists i.\ pc(i) = \mathsf{S1}]$. There, we have shown that this logic alone allows one to simulate the non-halting property of a two-counter machine, which immediately leads to undecidability [John et al. 2012][Theorem 6]. Curiously, this proof did not require the processes to communicate at all.

   Hence, it was crucial to find a fragment of LTL that is, on one hand, sufficiently expressive to capture the liveness properties of the threshold-guarded distributed algorithms, and, on the other hand, does not immediately lead to undecidability.

2. *Finding completeness thresholds for liveness.* It is well known that using the diameter bound in bounded model checking of liveness leads to incompleteness [Biere et al. 1999]. Hence, Kroening and Strichman [2003] and Kroening et al. [2011] computed recurrence diameters. However, in general the recurrence diameters are exponentially larger than the diameters, which makes them impractical for verification purposes.

3. *Reduction arguments for liveness.* In Chapters 4 and 5, the reordering of transitions had to be done in such a way that evaluation of the guards was not affected. When reordering transitions with respect to safety and liveness properties, one has to pay attention of not changing the atomic propositions in the intermediate configurations.

Main contributions. To address the first challenge, we introduce the logic $\mathsf{ELTL}_{\mathsf{FT}}$ that has a number of restrictions: (1) the only temporal operators are G (always) and F (eventually); (2) the temporal formulas can be only conjoined, no disjunctions are allowed; (3) the atomic propositions are restricted to linear arithmetic over the shared variables and the parameters, conjunctions $\bigwedge_{\ell \in Locs} \kappa[\ell] = 0$ and disjunctions $\bigvee_{\ell \in Locs} \kappa[\ell] \neq 0$. This logic existentially quantifies over the executions (as the prefix E in the name suggests), and thus allows us to express existence of an execution that would violate a safety or liveness property.

For the logic $\mathsf{ELTL}_{\mathsf{FT}}$, we show that one can construct a finite set of lasso templates that differ in the order, in which the temporal subformulas starting with F and G get satisfied. This result was inspired by the construction for $\mathsf{LTL}(\mathsf{F}, \mathsf{G})$ by Etessami et al. [2002]. We extend schemas with lasso-like schemas that end up in a loop, and show how one can combine the schemas for reachability with the lasso templates.

To address the second and the third challenges, we show that for the atomic propositions of $\mathsf{ELTL}_{\mathsf{FT}}$, one can reuse the schema construction from Chapter 5. To this end, we prove that there are a few processes whose transitions have to be explicitly tracked and the transitions of the other processes can be reordered as in the reachability case. As a result, we can repeat the reordering argument for reachability a few times, in order to construct schemas for safety and liveness.

Implementation and experiments. We have implemented this technique in the latest version of ByMC. These results allowed us to verify both safety and liveness of ten fault-tolerant distributed algorithms, including BOSCO by Song and van Renesse [2008], condition-based consensus by Mostéfaoui et al. [2003], consensus in one communication step by Brasileiro et al. [2001]. These experiments have successfully passed the artifact evaluation at POPL'17 [Konnov et al. 2017b].

Follow up results

In [Konnov et al. 2016b], we discussed the relation between the techniques of Parts I and II. We also gave an idea of how one would translate Promela models from Part I to threshold automata. This abstraction technique is implemented in Byzantine model checker (ByMC).

In our recent tool paper [Konnov and Widder 2018], we further discuss the difference between the modeling with threshold automata and explicit local message counters. We compare the efficiency of ByMC on the manually crafted threshold automata and the threshold automata that are automatically constructed by data abstraction. In most cases, the manually crafted threshold automata are significantly easier to verify. We also report on the parallel extension of the techniques introduced in Chapters 5 and 6.

In his master thesis, Kukovec [2016] investigated extensions of this result to other forms of threshold automata: guards with piecewise monotone functions, reversal-bounded, reversible, etc. While some of these extensions also have bounded diameters, in many cases parameterized reachability is undecidable. We also studied the link between counter systems of threshold automata and accelerated counter automata in [Kukovec et al. 2018].

Overview of Part III

Chapter 7: Synthesis of Distributed Algorithms with Parameterized Threshold Guards

The thresholds in the threshold-guarded algorithms often vary, depending on the assumptions about the computational model. The following thresholds are often met in the literature: $t+1$, $2t+1$, $n-t$, $\frac{n}{2}$, and $\frac{n-t}{2}$. It is folklore knowledge that some algorithms admit different combinations of thresholds. For instance, reliable broadcast by Srikanth and Toueg [1987b] has three such combinations: (1) $t+1$ and $2t+1$, (2) $t+1$ and $n-t$, and (3) $n-2t$ and $n-t$.

In this chapter, we automatically find the thresholds for the sketches of threshold automata, that is, threshold automata whose guards are partially specified. For instance, in the threshold automaton of BOSCO (Figure 2 on page 18), we replace the integer coefficients in the thresholds with unknown coefficients. The threshold τ_{D0} becomes $\mathbf{?}_a^{\tau_{\mathsf{D0}}} \cdot n + \mathbf{?}_b^{\tau_{\mathsf{D0}}} \cdot t + \mathbf{?}_c^{\tau_{\mathsf{D0}}}$, where $\mathbf{?}_a^{\tau_{\mathsf{D0}}}$, $\mathbf{?}_b^{\tau_{\mathsf{D0}}}$, and $\mathbf{?}_c^{\tau_{\mathsf{D0}}}$ are the unknowns that have to be found. Thus, the goal of the synthesis algorithm is to find the values for all the unknowns, so that the resulting system of threshold automata satisfies the expected safety and liveness properties.

Main contribution. To solve this synthesis problem, we integrate Byzantine model checker into the CEGIS loop [Alur et al. 2013]: A synthesis oracle gen-

erates the values of unknowns and feeds them to our model checker ByMC, and ByMC verifies the synthesized algorithm and feeds the counterexamples to the oracle. A naïve implementation of the CEGIS loop does not scale to the sophisticated algorithms such as BOSCO, as it keeps enumerating the vectors of unknowns without analyzing the counterexamples. Hence, we introduce an approach to generalize the counterexamples that are generated by the model checker, in order to exclude large sets of executions. The key to this generalization is in using the schemas. When the model checker finds a counterexample, it analyzes the schema that was used to produce the counterexample and generates an SMT query that excludes many values of unknowns that would produce similar counterexamples. This synthesis technique scales up to our benchmarks.

As the synthesis oracle draws values of unknowns from an infinite vector space, the synthesis loop is not guaranteed to terminate. To bound the search space, we restrict the thresholds to what we call "sane guards". Sane guards compare the values of message counters to values from the interval $[0, n]$, which is quite natural for threshold-guarded algorithms — indeed, the message counters cannot go below 0 or above n. For the sane guards, we show that there is a bounded search space, which guarantees termination of the synthesis loop.

Implementation and experiments. We have synthesized reliable broadcast by Srikanth and Toueg [1987b], hybrid broadcast by Widder and Schmid [2007], and BOSCO by Song and van Renesse [2008]. Interestingly, we have shown that the resilience conditions of BOSCO are tight, and there are no other combinations of threshold guards. We have also synthesized variations of the algorithms that satisfy variations of the original specifications.

Overview of Part IV

Chapter 8: Parameterized Systems in BIP: Design and Model Checking

In general, parameterized model checking is undecidable [Apt and Kozen 1986, Suzuki 1988]. Thus, the research efforts in the field focus on finding classes of parameterized protocols that have (semi-) decidable properties. These classes are given in terms of "computational models", that is, mathematical definitions of concurrency, communication, etc., which vary in subtle details. The key

features of a particular computational model often become apparent only after analyzing the respective decidability proof.

When writing the survey book on parameterized verification [Bloem et al. 2015], we noticed that the parameterized model checking techniques share the same shortcomings:

1. To apply a technique, one has to invest a lot of manual effort, as the system architecture and process behavior must be captured in the formal language specific to the technique.
2. Given a distributed algorithm or the architecture of a distributed system, it is often hard to tell, whether this specific instance belongs to a decidable fragment, and, if so, which technique one should use.
3. It is hard to compare different techniques, as they are presented in different computational models.

In the non-parameterized case, many forms of communication can be captured in the Behavior-Interaction-Priority (BIP), which was introduced by Basu et al. [2011] to encompass various architectural styles. This framework builds upon the notion of an *interaction*, which identifies the components that can communicate within the interaction. The set of possible system interactions is defined in BIP as a model of a formula in propositional interaction logic.

Main contributions. In this chapter, we extend the propositional interaction logic to a parameterized version, which we call *first-order interaction logic* (FOIL). Using FOIL, we formalize the following parameterized architectures: cliques communicating via pairwise rendezvous, cliques communicating via synchronous broadcasts, and token-passing rings.

We show how to encode FOIL formulas for those architectures in SMT. Hence, given a parameterized BIP design, our technique automatically finds the architecture, to which the design belongs. This gives us a way of systematically integrating many parameterized model checking techniques in a single tool.

Experiments. We have implemented a prototype tool that takes a parameterized BIP design at its input and checks, whether it fits the computational models of the classical techniques: VASS techniques for star architectures and cliques by German and Sistla [1992], cut-off techniques for token rings by Emerson and

Namjoshi [1995], well-structured transition systems techniques for broadcast systems by Abdulla et al. [1996]. This tool successfully found the architectures of the following classical protocols: Milner's scheduler, semaphore implementation with rendezvous, and barrier synchronization.

Individual contribution to the papers

The results presented in this cumulative habilitation thesis are based on joint work of several co-authors that was done in an equally distributed joint collaboration, which makes it hard to define individual roles of each co-author. To reflect this, as is customary in the field, in most cases, the order of the authors' names is alphabetical. When it is not the case, the co-authors wished to emphasize more significant contribution of the first authors.

The authors of [Konnov et al. 2016a] decided to change the order of the authors and put my name and the names of Tomer Kotek and Qiang Wang first to reflect that fact that a large body of the work was done during Wang's internship at TU Wien. However, all co-authors contributed to developing the theoretical framework, writing the paper, etc.

For the papers John et al. [2013c], Konnov et al. [2017a;b;c], Lazic et al. [2017], apart from contributing to the theoretical framework and the proofs, as the author of ByMC, I implemented the techniques in this tool. I believe that this does not lessen the contributions of my co-authors.

References

P. A. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay. General decidability theorems for infinite-state systems. In *LICS*, 1996.

Ittai Abraham, Dahlia Malkhi, et al. The blockchain consensus layer and BFT. *Bulletin of EATCS*, 3(123), 2017.

Francesco Alberti, Silvio Ghilardi, Elena Pagani, Silvio Ranise, and Gian Paolo Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2):29–61, 2012.

Francesco Alberti, Silvio Ghilardi, Andrea Orsini, and Elena Pagani. Counter abstractions in model checking of distributed broadcast algorithms: Some case studies. In *31st Italian Conf. on Comp. Logic*, pages 102–117, 2016.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.

Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger. Parameterized model checking of synchronous distributed algorithms by abstraction. In *VMCAI*, volume 10747 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2018.

K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.

Hagit Attiya and Jennifer Welch. *Distributed Computing.* Wiley, 2nd edition, 2004.

Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20:20–20:32, July 2014.

Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with SLAM. *CACM*, 54(7):68–76, 2011.

A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *Software, IEEE*, 2011.

Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *QEST*, pages 125–126, 2006.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207, 1999.

Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

Aaron R. Bradley. IC3 and beyond: Incremental, inductive verification. In *CAV*, page 4, 2012.

Francisco Vilar Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *PaCT*, volume 2127 of *LNCS*, pages 42–50, 2001.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *LICS*, pages 428–439, 1990.

Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

Kārlis Čerāns. Decidability of bisimulation equivalences for parallel timer processes. In *CAV*, volume 663 of *LNCS*, pages 302–315, 1993.

Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, March 1996.

Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407. ACM, 2007.

Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71, 1981.

Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *VMCAI*, volume 2937 of *LNCS*, pages 85–96, 2004.

Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 337–340. 2008.

Giorgio Delzanno, Michele Tatarek, and Riccardo Traverso. Model checking paxos in spin. In *Int. Symposium on Games, Automata, Logics and Formal Verification*, pages 131–146, 2014.

Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2014.

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415. ACM, 2016.

E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.

Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.

Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

Dana Fisman, Orna Kupferman, and Yoad Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, volume 4963 of *LNCS*, pages 315–331. Springer, 2008.

Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.

A. Geist. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum*, 10, 2016.

Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.

Annu Gmeiner, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *Formal Methods for Executable Software Models*, LNCS, pages 122–171. Springer, 2014.

Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, volume 531 of *LNCS*, pages 176–185, 1990.

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83, 1997.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS*, pages 441–444, 2006.

Gerard Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Counter attack on Byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms. *arXiv CoRR*, abs/1210.3846, 2012.

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Brief announcement: parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *PODC*, pages 119–121, 2013a.

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *SPIN*, volume 7976 of *LNCS*, pages 209–226, 2013b.

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013c.

Igor Konnov and Josef Widder. ByMC: Byzantine model checker. In *ISoLA 2018, Part III*, LNCS, pages 327–342, 2018. doi: 10.1007/978-3-030-03424-5_22.

Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In *CONCUR*, volume 8704 of *LNCS*, pages 125–140, 2014.

Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

Igor Konnov, Tomer Kotek, Qiang Wang, Helmut Veith, Simon Bliudze, and Joseph Sifakis. Parameterized Systems in BIP: Design and Model Checking. In *CONCUR 2016*, volume 59 of *LIPIcs*, pages 30:1–30:16, 2016a.

Igor Konnov, Helmut Veith, and Josef Widder. What you always wanted to know about model checking of fault-tolerant distributed algorithms. In *PSI 2015, in Memory of Helmut Veith, Revised Selected Papers*, volume 9609 of *LNCS*, pages 6–21. Springer, 2016b.

Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. Para^2: Parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design*, 51(2): 270–307, 2017a.

Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734, 2017b.

Igor V. Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Inf. Comput.*, 252:95–109, 2017c.

Igor V. Konnov, Josef Widder, Francesco Spegni, and Luca Spalazzi. Accuracy of message counting abstraction in fault-tolerant distributed algorithms. In *VMCAI*, pages 347–366, 2017d.

Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *VMCAI*, volume 2575 of *LNCS*, pages 298–309, 2003.

Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear completeness thresholds for bounded model checking. In *CAV*, volume 6806 of *LNCS*, pages 557–572, 2011.

Jure Kukovec. Generalizing threshold automata for reachability in parameterized systems. Master's thesis, University of Ljubljana, 2016. URL: `http://forsyte.at/wp-content/uploads/Kukovec-27142109-2016.pdf`.

Jure Kukovec, Igor Konnov, and Josef Widder. Reachability in parameterized systems: All flavors of threshold automata. In *CONCUR 2018*, pages 19:1–19:17, 2018.

Marta Z. Kwiatkowska and Gethin Norman. Verifying randomized byzantine agreement. In *FORTE*, pages 194–209, 2002.

Marta Z. Kwiatkowska, Gethin Norman, and Roberto Segala. Automated verification of a randomized distributed consensus protocol using cadence SMV and PRISM. In *CAV*, pages 194–206, 2001.

Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

Leslie Lamport. "Sometime" is sometimes "not never" - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.

Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.

Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc., 2002.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS*, volume 95 of *LIPIcs*, pages 32:1–32:20, 2017.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16, Revised Selected Papers*, pages 348–370, 2010.

Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

Nancy A. Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, 1989.

Ognjen Marić, Christoph Sprenger, and David A. Basin. Cutoff Bounds for Consensus Algorithms. In *CAV*, pages 217–237, 2017.

Kenneth L. McMillan. Modular specification and verification of a cache-coherent interface. In *FMCAD*, pages 109–116. IEEE, 2016.

Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *DISC*, pages 49–63, 1999.

Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

T. Noguchi, T. Tsuchiya, and T. Kikuno. Safety verification of asynchronous consensus algorithms with model checking. In *Dependable Computing (PRDC)*, pages 80–88, 2012.

Diego Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford U., 2014.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–320, 2014.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL*, 1 (OOPSLA):108:1–108:31, 2017.

Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J.ACM*, 27(2):228–234, 1980.

Doron Peled. All from one, one for all: on model checking using representatives. In *CAV*, volume 697 of *LNCS*, pages 409–423, 1993.

Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with $(0,1,\infty)$- counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 93–111. 2002.

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Proceedings*, pages 337–351, 1982.

Alexander Randall, 5th. Q&A: A lost interview with ENIAC co-inventor J. Presper Eckert, February 2006. URL `https://www.computerworld.com/article/2561813/computer-hardware/q-a--a-lost-interview-with-eniac-co-inventor-j--presper-eckert.html`. [Online; retrieved 20-September-2018].

Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987a.

T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987b.

Ichiro Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.

Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *FMSD*, 18:25–68, 2001.

Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Dist. Comp.*, 23(5–6):341–358, 2011.

Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.

Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Dist. Comp.*, 20 (2):115–140, 2007.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

Part I

Modeling of Fault-Tolerant Distributed Algorithms and Model Checking by Abstraction

Chapter 1

Towards modeling and model checking fault-tolerant distributed algorithms

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards modeling and model checking fault-tolerant distributed algorithms. SPIN, vol. 7976 of *LNCS*, pp. 209–226, 2013.

DOI: http://dx.doi.org/10.1007/978-3-642-39176-7_14

Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms*

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder

Vienna University of Technology (TU Wien)

Abstract. Fault-tolerant distributed algorithms are central for building reliable, spatially distributed systems. In order to ensure that these algorithms actually make systems more reliable, we must ensure that these algorithms are actually correct. Unfortunately, model checking state-of-the-art fault-tolerant distributed algorithms (such as Paxos) is currently out of reach except for very small systems.

In order to be eventually able to automatically verify such fault-tolerant distributed algorithms also for larger systems, several problems have to be addressed. In this paper, we consider modeling and verification of fault-tolerant algorithms that basically only contain threshold guards to control the flow of the algorithm. As threshold guards are widely used in fault-tolerant distributed algorithms (and also in Paxos), efficient methods to handle them bring us closer to the above mentioned goal.

As a case study we use the reliable broadcasting algorithm by Srikanth and Toueg that tolerates even Byzantine faults. We show how one can model this basic fault-tolerant distributed algorithm in PROMELA such that safety and liveness properties can be efficiently verified in SPIN. We provide experimental data also for other distributed algorithms.

1 Introduction

Even formally verified computer systems are subject to power outages, electrical wear-out, bit-flips in memory due to ionizing particle hits, etc., which may easily cause system failures. Replication is a classic approach to ensure that a computer system is fault-tolerant, i.e., continues to correctly perform its task even if some components fail. The basic idea is to have multiple computers instead of a single one (that would constitute a single point of failure), and ensure that the replicated computers coordinate, and for instance in the case of replicated databases, store the same information. Ensuring that all computers agree on the same information is non-trivial due to several sources of non-determinism, namely, faults, uncertain message delays, and asynchronous computation steps.

To address these issues, fault-tolerant distributed algorithms for state machine replication were introduced many years ago [33]. As they are designed to

* Supported by the Austrian National Research Network S11403 and S11405 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED, ICT12-059, and VRG11-005.

E. Bartocci and C.R. Ramakrishnan (Eds.): SPIN 2013, LNCS 7976, pp. 209–226, 2013.
© Springer-Verlag Berlin Heidelberg 2013

increase the reliability of computing systems, it is crucial that these algorithms are indeed correct, i.e., satisfy their specifications. Due to the various sources of non-determinism, however, it is very easy to make mistakes in the correctness arguments for fault-tolerant distributed algorithms. As a consequence, they are very natural candidates for model checking. Still, model checking fault-tolerant distributed algorithms is particularly challenging due to the following reasons:

(i) Due to their inherent concurrency and the many sources of non-determinism, fault-tolerant distributed algorithms suffer from combinatorial explosion in the state-space, and in the number of behaviors. Moreover, distributed algorithms usually involve parameters such as the system size n and the maximum number of faulty components t.
(ii) Correctness and even solvability of problems like distributed agreement depend critically upon assumptions on the environment, in particular, degree of concurrency, message delays, and failure models; e.g., guaranteeing correct execution is impossible if there is no restriction on the number of faulty components in the system and/or the way how they may fail.
(iii) There is no commonly agreed-upon distributed computing model, but rather many variants, which differ in (sometimes subtle) details such as atomicity of a computing step. Moreover, distributed algorithms are usually described in pseudocode, typically using different (alas unspecified) pseudocode languages, which obfuscates the relation to the underlying computing model.

A central and important goal of our recent work is hence to initiate a systematic study of distributed algorithms from a verification point of view, in a way that does not betray the fundamentals of distributed algorithms. Experience tells that this has not always been observed in the past: The famous bakery algorithm [22] is probably the most striking example from the literature where wrong specifications have been verified or wrong semantics have been considered: Many papers in formal methods have verified the correctness of the bakery algorithm as an evidence for their practical applicability. Viewed from a distributed algorithms perspective, however, most of these papers missed the fact that the algorithm does not require atomic registers but rather safe registers only [23] — a subtle detail that is admittedly difficult to extract from the distributed algorithms literature for non-experts. Still, compared to state-of-the-art fault-tolerant distributed algorithms — and even the algorithms considered in this paper — the bakery algorithm rests on a quite simple computational model, which shows the need for a structured approach to handle distributed algorithms.

Contributions. In this paper, we present a structured approach for modeling an important family of fault-tolerant distributed algorithms, namely, threshold-guarded distributed algorithms discussed in Section 2. As threshold-guarded commands are omnipresent in this domain, our work is an important step towards the goal of verifying state-of-the-art fault-tolerant distributed algorithms. In Section 3, we obtain models of distributed algorithms expressed in slightly extended PROMELA [20] to capture the notions required to fully express fault-tolerant distributed algorithms and their environments, including resilience

conditions involving parameters like n and t, fairness conditions, and atomicity assumptions. This formalization allows us to (i) instantiate system instances for different system sizes in order to perform explicit state model checking using Spin as discussed in Section 4, and (ii) build a basis for our parameterized model checking technique based on parametric interval abstraction discussed in [21].

Using our approach, we can already formalize and model check several basic fault-tolerant distributed algorithms for fixed parameters, i.e., numbers of processes and faults. These algorithms include several variants of the classic asynchronous broadcasting algorithm from [34] under various fault assumptions, the broadcasting algorithm from [6] tolerating Byzantine faults, the classic broadcasting algorithm found, e.g., in [9], that tolerates crash faults, as well as a condition-based consensus algorithm [27] that also tolerates crash faults.

This captures the most interesting problems that are solvable [16] by distributed algorithms running in a purely asynchronous environment with faults. Our verification results build a corner stone for the verification of more advanced fault-tolerant distributed algorithms [13,9,26,37,10,18]. These algorithms use threshold-guarded commands as a building block, yet contain other features that call for additional model checking techniques.

2 Threshold-Guarded Distributed Algorithms

Processes, which constitute the distributed algorithms we consider, exchange messages, and change their state predominantly based on the received messages. In addition to the standard execution of actions, which are guarded by some predicate on the local state, most basic distributed algorithms (cf. [24,3]) add existentially or universally guarded commands involving received messages:

```
if received <m>
   from some process
then action(m);
```

(a) existential guard

```
if received <m>
   from all processes
then action(m);
```

(b) universal guard

Depending on the content of the message <m>, the function `action` performs a local computation, and possibly sends messages to one or more processes. Such constructs can be found, e.g., in (non-fault-tolerant) distributed algorithms for constructing spanning trees, flooding, mutual exclusion, or network synchronization [24]. Understanding and analyzing such distributed algorithms is already far from being trivial, which is due to the partial information on the global state present in the local state of a process. However, faults add another source of nondeterminism. In order to shed some light on the difficulties facing a distributed algorithm in the presence of faults, consider Byzantine faults [28], which allow a faulty process to behave arbitrarily: Faulty processes may fail to send messages, send messages with erroneous values, or even send conflicting information to different processes. In addition, faulty processes may even collaborate in order to increase their adverse power.

Fault-tolerant distributed algorithms work in the presence of such faults and provide some "higher level" service: In case of distributed agreement (or consensus), e.g., this service is that all non-faulty processes compute the same result even if some processes fail. Fault-tolerant distributed algorithms are hence used for increasing the system-level reliability of distributed systems [30].

If one tries to build such a fault-tolerant distributed algorithm using the construct of Example (a) in the presence of Byzantine faults, the (local state of the) receiver process would be corrupted if the received message `<m>` originates in a faulty process. A faulty process could hence contaminate a correct process. On the other hand, if one tried to use the construct of Example (b), a correct process would wait forever (starve) when a faulty process omits to send the required message. To overcome those problems, fault-tolerant distributed algorithms typically require assumptions on the maximum number of faults, and employ suitable thresholds for the number of messages which can be expected to be received by correct processes. Assuming that the system consists of n processes among which at most t may be faulty, *threshold-guarded commands* such as the following are typically used in fault-tolerant distributed algorithms:

```
if received <m> from n-t distinct processes
then action(m);
```

Assuming that thresholds are functions of the parameters n and t, threshold guards are a just generalization of quantified guards as given in Examples (a) and (b): In the above command, a process waits to receive $n - t$ messages from distinct processes. As there are at least $n - t$ correct processes, the guard cannot be blocked by faulty processes, which avoids the problems of Example (b). In the distributed algorithms literature, one finds a variety of different thresholds: Typical numbers are $\lceil n/2+1 \rceil$ (for majority [13,27]), $t+1$ (to wait for a message from at least one correct process [34,13]), or $n - t$ (in the Byzantine case [34,2] to wait for at least $t + 1$ messages from correct processes, provided $n > 3t$).

In the setting of Byzantine fault tolerance, it is important to note that the use of threshold-guarded commands implicitly rests on the assumption that a receiver can distinguish messages from different senders. This can be achieved, e.g., by using point-to-point links between processes or by message authentication. What is important here is that Byzantine faulty processes are only allowed to exercise control on their own messages and computations, but not on the messages sent by other processes and the computation of other processes.

Reliable Broadcast and Related Specifications. The specifications considered in the area of fault tolerance differ from more classic areas, such as concurrent systems where dining philosophers and mutual exclusion are central problems. For the latter, one is typically interested in local properties, e.g., if a philosopher i is hungry, then i eventually eats. Intuitively, dining philosophers requires us to trace indexed processes along a computation, e.g., $\forall i.\ \mathbf{G}\,(\text{hungry}_i \rightarrow (\mathbf{F}\,\text{eating}_i))$, and thus to employ *indexed* temporal logics for specifications [7,11,12,14].

In contrast, fault-tolerant distributed algorithms are typically used to achieve *global* properties. Reliable broadcast is an ongoing "system service" with the

following informal specification: Each process i may invoke a primitive called broadcast by calling $bcast(i, m)$, where m is a unique message content. Processes may deliver a message by invoking $accept(i, m)$ for different process and message pairs (i, m). The goal is that all correct processes invoke $accept(i, m)$ for the same set of (i, m) pairs, under some additional constraints: all messages broadcast by correct processes must be accepted by all correct processes, and $accept(i, m)$ may not be invoked, unless i is faulty or i invoked $bcast(i, m)$. Our case study is to verify that the algorithm from [34] implements these primitives on top of point-to-point channels, in the presence of Byzantine faults. In [34], the instances for different (i, m) pairs do not interfere. Therefore, we will not consider i and m. Rather, we distinguish the different kinds of invocations of $bcast(i, m)$ that may occur, e.g., the cases where the invoking process is faulty or correct. Depending on the initial state, we then have to check whether every/no correct process accepts. To capture this kind of properties, we have to trace only existentially or universally quantified properties, e.g., a part of the broadcast specification (relay) [34] states that if some correct process accepts a message, then all (correct) processes accept the message, that is, $\mathbf{G}\,((\exists i.\ \text{accept}_i) \rightarrow \mathbf{F}\,(\forall j.\ \text{accept}_j))$.

We are therefore considering a temporal logic where the *quantification over processes is restricted to propositional formulas.* We will need two kinds of quantified propositional formulas that consider (i) the finite control state modeled as a single status variable sv, and (ii) the possible unbounded data. We introduce the set AP_{SV} that contains propositions that capture comparison against some status value Z from the set of all control states, i.e., $[\forall i.\ sv_i = Z]$ and $[\exists i.\ sv_i = Z]$.

This allows us to express specifications of distributed algorithms. To express the mentioned relay property, we identify the status values where a process has accepted the message. We may quantify over all processes as we only explicitly model those processes that follow their code, that is, correct or benign faulty processes. More severe faults that are unrestricted in their internal behavior (e.g., Byzantine faults) are modeled via non-determinism in message passing. For a detailed discussion see Section 3.

In order to express comparison of data variables, we add a set of atomic propositions AP_D that capture comparison of data variables (integers) x, y, and constant c; AP_D consists of propositions of the form $[\exists i.\ x_i + c < y_i]$.

The labeling function of a system instance is then defined naturally as disjunction or conjunction over all process indices; cf. [21] for complete definitions.

Given an $\mathsf{LTL} \setminus \mathsf{X}$ formula ψ over AP_D expressing justice [29], an $\mathsf{LTL} \setminus \mathsf{X}$ specification φ over AP_{SV}, a process description P in Promela, and the number of (correct) processes N, the problem is to verify whether

$$\underbrace{P \parallel P \parallel \cdots \parallel P}_{N times} \models \psi \rightarrow \varphi.$$

3 Threshold-Guarded Distributed Algorithms in Promela

Algorithm 1 is our case study for which we also provide a complete Promela implementation later in Figure 4. To explain how we obtain this implementation,

Algorithm 1. Core logic of the broadcasting algorithm from [34]

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

we proceed in three steps where we first discuss asynchronous distributed algorithms in general, then explain our encoding of message passing for threshold-guarded fault-tolerant distributed algorithms. Algorithm 1 belongs to this class, as it does not distinguish messages according to their senders, but just counts received messages, and performs state transitions depending on the number of received messages; e.g., line 7. Finally we encode the control flow of Algorithm 1. The rationale of the modeling decisions are that the resulting Promela model (i) captures the assumptions of distributed algorithms adequately, and (ii) allows for efficient verification either using explicit state enumeration (as discussed in this paper) or by abstraction as discussed in [21]. After discussing the modeling of distributed algorithms, we will provide the specifications in Section 3.4.

3.1 Computational Model for Asynchronous Distributed Algorithms

We recall the standard assumptions for asynchronous distributed algorithms. A system consists of n processes, out of which at most t may be faulty. When considering a fixed computation, we denote by f the actual number of faulty processes. Note that f is not "known" to the processes. It is assumed that $n > 3t \wedge f \le t \wedge t > 0$. Correct processes follow the algorithm, in that they take steps that correspond to the algorithm. Between every pair of processes, there is a bidirectional link over which messages are exchanged. A link contains two message buffers, each being the receive buffer of one of the incident processes.

A step of a correct process is *atomic* and consists of the following three parts. (i) The process possibly receives a message. A process is not forced to receive a message even if there is one in its buffer [16]. (ii) Then, it performs a state transition depending on its current state and the (possibly) received message. (iii) Finally, a process may send at most one message to each process, that is, it puts a message in the buffers of the other processes.

Computations are asynchronous in that the steps can be arbitrarily interleaved, provided that each correct process takes an infinite number of steps.

(Algorithm 1 has runs that never accept and are infinite. Conceptually, the standard model requires that processes executing terminating algorithms loop forever in terminal states [24].) Moreover, if a message m is put into process p's buffer, and p is correct, then m is eventually received. This property is called *reliable communication.*

From the above discussion we observe that buffers are required to be unbounded, and thus sending is non-blocking. Further, receiving is non-blocking even if no message has been sent to the process. If we assume that for each message type, each correct process sends at most one message in each run (as in Algorithm 1), non-blocking send can in principle natively be encoded in Promela using message channels. In principle, non-blocking receive also can be implemented in Promela, but it is not a basic construct. We discuss the modeling of message passing in more detail in Section 3.2.

Fault Types. In our case study Algorithm 1 we consider *Byzantine* faults, that is, faulty processes are not restricted, except that they have no influence on the buffers of links to which they are not incident. Below we also consider restricted failure classes: *omission faults* follow the algorithm but may fail to send some messages, *crash faults* follow the algorithm but may prematurely stop running. Finally, *symmetric faults* need not follow the algorithm, but if they send messages, they send them to all processes. (The latter restriction does not apply to Byzantine faults which may send conflicting information to different processes).

Verification Goal. Recall that there is a condition on the parameters n, t, and f, namely, $n > 3t \wedge f \leq t \wedge t > 0$. As these parameters do not change during a run, they can be encoded as constants in Promela. The verification problem for a distributed algorithm with fixed n and t is then the composition of model checking problems that differ in the actual value of f (satisfying $f \leq t$).

3.2 Efficient Encoding of Message Passing

In threshold-guarded distributed algorithms, the processes (i) count how many messages of the same type they have received from *distinct* processes, and change their states depending on this number, (ii) always send to *all* processes (including the process itself), and (iii) send messages only for a fixed number of types (only messages of type ⟨echo⟩ are sent in Algorithm 1).

Fault-Free Communication. We discuss in the following that one can model such algorithms in a way that is more efficient in comparison to a straightforward implementation with Promela channels. In our final modeling we have an approach that captures both message passing and the influence of faults on correct processes. However, in order to not clutter the presentation, we start our discussion by considering communication between correct processes only (i.e., $f = 0$), and add faults later in this section.

In the following code examples we show a straightforward way to implement "received ⟨echo⟩ from at least x distinct processes" and "send ⟨echo⟩ to all"

using PROMELA channels: We declare an array p2p of n^2 channels, one per pair of processes, and then we declare an array rx to record that at most one ⟨echo⟩ message from a process j is received by a process i:

```
mtype = { ECHO }; /* one message type */
chan p2p[NxN] = [1] of { mtype }; /* channels of capacity 1 */
bit  rx[NxN]; /* a bit map to implement "distinct" */
active[N] proctype STBcastChan() {
  int i, nrcvd = 0; /* nr. of echoes */
```

Then, the receive code iterates over n channels: for non-empty channels it receives an ⟨echo⟩ message or not, and empty channels are skipped; if a message is received, the channel is marked in rx:

```
i = 0; do
  :: (i < N) && nempty(p2p[i * N + _pid]) ->
    p2p[i * N + _pid]?ECHO; /* retrieve a message */
    if
      :: !rx[i * N + _pid] ->
        rx[i * N + _pid] = 1; /* mark the channel */
        nrcvd++; break; /* receive at most one message */
      :: rx[i * N + _pid];   /* ignore duplicates */
    fi; i++;
  :: (i < N) ->
    i++;   /* channel is empty or postpone reception */
  :: i == N -> break;
od
```

Finally, the sending code also iterates over n channels and sends on each:

```
for (i : 1 .. N) { p2p[_pid * N + i]!ECHO; }
```

Recall that threshold-guarded algorithms have specific constraints: messages from all processes are processed uniformly; every message is carrying only a message type without a process identifier; each process sends a message to all processes in no particular order. This suggests a simpler modeling solution. Instead of using message passing directly, we keep only the numbers of sent and received messages in integer variables:

```
int nsnt; /* one shared variable per a message type */
active[N] proctype STBcast() {
  int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes */
  ...
step: atomic {
    if /* receive one more echo */
      :: (next_nrcvd < nsnt) ->
        next_nrcvd = nrcvd + 1;
      :: next_nrcvd = nrcvd; /* or nothing */
    fi;
    ...
    nsnt++; /* send echo to all */
  }
```

```
active[F] proctype Byz() {
step: atomic {
  i = 0; do
   :: i < N -> sendTo(i); i++;
   :: i < N -> i++; /* some */
   :: i == N -> break;
  od
 }; goto step;
}

active[F] proctype Omit() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 if :: correctCodeSendsAll ->
   i = 0; do
    :: i < N -> sendTo(i); i++;
    :: i < N -> i++; /* omit */
    :: i == N -> break;
   od
   :: skip;
 fi
 }; goto step;
}
```

```
active[F] proctype Symm() {
step: atomic {
  if
   :: /* send all */
     for (i : 1 .. N)
     { sendTo(i); }
   :: skip; /* or none */
  fi
 }; goto step;
}

active[F] proctype Clean() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 /* send as a correct one */
 };
 if
    :: goto step;
    :: goto crash;
 fi;
crash:
}
```

Fig. 1. Modeling faulty processes explicitly: Byzantine (Byz), symmetric (Symm), omission (Omit), and clean crashes (Clean)

As one process step is executed atomically (indivisibly), concurrent reads and updates of *nsnt* are not a concern to us. Note that the presented code is based on the assumption that each correct process sends at most one message. We show how to enforce this assumption when discussing the control flow of our implementation of Algorithm 1 in Section 3.3.

Recall that in asynchronous distributed systems one assumes communication fairness, that is, every message sent is eventually received. The statement $\exists i.\ rcvd_i < nsnt_i$ describes a global state where messages are still in transit. It follows that a formula ψ defined by

$$\mathsf{G}\,\mathsf{F} \neg [\exists i.\ rcvd_i < nsnt_i] \qquad \text{(RelComm)}$$

states that the system periodically delivers all messages sent by (correct) processes. We are thus going to add such fairness requirements to our specifications.

Faulty Processes. In Figure 1 we show how one can model the different types of faults discussed above using channels. The implementations are direct consequences of the fault description given in Section 3.1. Figure 2 shows how the impact of faults on processes following the algorithm can be implemented in the shared memory implementation of message passing. Note that in contrast to

```
/* N > 3T ∧ T ≥ F ≥ 0 */
active[N-F] proctype ByzI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt + F)
    -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N > 2T ∧ T ≥ Fp ≥ Fs ≥ 0 */
active[N-Fp] proctype SymmI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt + Fs)
    -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N > 2T ∧ T ≥ F ≥ 0 */
active[N] proctype OmitI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt) ->
    next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N ≥ T ∧ T ≥ Fc ≥ Fnc ≥ 0 */
active[N] proctype CleanI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt - Fnc)
     -> next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

Fig. 2. Modeling the effect of faults on correct processes: Byzantine (ByzI), symmetric (SymmI), omission (OmitI), and clean crashes (CleanI)

Figure 1, the processes in Figure 2 are *not* the faulty ones, but correct ones whose variable `next_nrcvd` is subject to non-deterministic updates that correspond to the impact of faulty process. For instance, in the Byzantine case, in addition to the messages sent by correct processes, a process can receive up to f messages more. This is expressed by the condition `(next_nrcvd < nsnt + F)`.

For Byzantine and symmetric faults we only model correct processes explicitly. Thus, we specify that there are `N-F` copies of the process. Moreover, we can use Property (RelComm) to model reliable communication. Omission and crash faults, however, we model explicitly, so that we have `N` copies of processes. Without going into too much detail, the impact of faulty processes is modeled by relaxed fairness requirements: as some messages sent by these f faulty processes may not be received, this induces less strict communication fairness:

$$\mathsf{G}\,\mathsf{F}\,\neg\,[\exists i.\ rcvd_i + f < nsnt_i]$$

By similar adaptations one models, e.g., corrupted communication (e.g., due to faulty links) [31], or hybrid fault models [4] that contain different fault scenarios.

Figure 3 compares the number of states and memory consumption when modeling message passing using both solutions. We ran SPIN to perform exhaustive

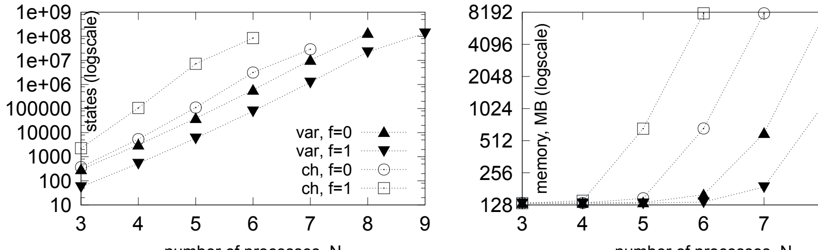


Fig. 3. Visited states (left) and memory usage (right) when modeling message passing with channels (ch) or shared variables (var). The faults are in effect only when $f > 0$. Ran with SAFETY, COLLAPSE, COMP, and 8GB of memory.

state enumeration on the encoding of Algorithm 1 (discussed in the next section). As one sees, the model with explicit channels and faulty processes ran out of memory on *six* processes, whereas the shared memory model did so only with *nine* processes. Moreover, the latter scales better in the presence of faults, while the former degrades with faults. This leads us to use the shared memory encoding based on *nsnt* variables.

3.3 Encoding the Control Flow

Recall Algorithm 1, which is written in typical pseudocode found in the distributed algorithms literature. The lines `3-8` describe one step of the algorithm. Receiving messages is implicit and performed before line `3`, and the actual sending of messages is deferred to the end, and is performed after line `8`.

We encoded the algorithm in Figure 4 using custom PROMELA extensions to express notions of fault-tolerant distributed algorithms. The extensions are required to express a parameterized model checking problem, and are used by our tool that implements the abstraction methods introduced in [21]. These extensions are only syntactic sugar when the parameters are fixed: `symbolic` is used to declare parameters, and `assume` is used to impose resilience conditions on them (but is ignored in explicit state model checking). Declarations `atomic <var> = all (...)` are a shorthand for declaring atomic propositions that are unfolded into conjunctions over all processes (similarly for `some`). Also we allow expressions over parameters in the argument of `active`.

In the encoding in Figure 4, the whole step is captured within an atomic block (lines 20–42). As usual for fault-tolerant algorithms, this block has three logical parts: the receive part (lines 21–24), the computation part (lines 25–32), and the sending part (lines 33–38). As we have already discussed the encoding of message passing above, it remains to discuss the control flow of the algorithm.

Control State of the Algorithm. Apart from receiving and sending messages, Algorithm 1 refers to several facts about the current control state of a process: "sent ⟨*echo*⟩ before", "if v_i", and "$accept_i \leftarrow$ TRUE". We capture all possible

```
 1 symbolic int N, T, F;  /* parameters  */
 2 /* the resilience condition */
 3 assume(N > 3 * T && T >= 1 && 0 <= F && F <= T);
 4 int nsnt;  /* number of echoes sent by correct processes */
 5 /* quantified atomic propositions */
 6 atomic prec_unforg = all(STBcast:sv == V0);
 7 atomic prec_corr = all(STBcast:sv == V1);
 8 atomic prec_init = all(STBcast@step);
 9 atomic ex_acc = some(STBcast:sv == AC);
10 atomic all_acc = all(STBcast:sv == AC);
11 atomic in_transit = some(STBcast:nrcvd < nsnt);
12
13 active[N - F] proctype STBcast() {
14   byte sv, next_sv;          /* status of the algorithm */
15   int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes received */
16   if  /* initialize */
17     :: sv = V0; /* v_i = FALSE */
18     :: sv = V1; /* v_i = TRUE */
19   fi;
20 step: atomic { /* an indivisible step */
21     if /* receive one more echo (up to nsnt + F) */
22       :: (next_nrcvd < nsnt + F) -> next_nrcvd = nrcvd + 1;
23       :: next_nrcvd = nrcvd; /* or nothing */
24     fi;
25     if /* compute */
26       :: (next_nrcvd >= N - T) ->
27         next_sv = AC; /* accept_i = TRUE */
28       :: (next_nrcvd < N - T && sv == V1
29             || next_nrcvd >= T + 1) ->
30         next_sv = SE; /* remember that <echo> is sent */
31       :: else -> next_sv = sv; /* keep the status */
32     fi;
33     if /* send */
34       :: (sv == V0 || sv == V1)
35            && (next_sv == SE || next_sv == AC) ->
36         nsnt++; /* send <echo>  */
37       :: else;  /* send nothing */
38     fi;
39     /* update local variables and reset scratch variables */
40     sv = next_sv; nrcvd = next_nrcvd;
41     next_sv = 0; next_nrcvd = 0;
42     } goto step;
43 }
44 /* LTL-X formulas */
45 ltl fairness { []<>(!in_transit) } /* added to other formulas */
46 ltl relay { [](ex_acc -> <>all_acc) }
47 ltl corr { []((prec_init && prec_corr) -> <>(ex_acc)) }
48 ltl unforg { []((prec_init && prec_unforg) -> []!ex_acc) }
```

Fig. 4. Encoding of Algorithm 1 in PROMELA with symbolic extensions

control states in a finite set SV. For instance, for Algorithm 1 one can collect the set $SV = \{\text{V0}, \text{V1}, \text{SE}, \text{AC}\}$, where:

- V0 corresponds to $v_i = \text{FALSE}$, $\text{accept}_i = \text{FALSE}$ and ⟨echo⟩ is not sent.
- V1 corresponds to $v_i = \text{TRUE}$, $\text{accept}_i = \text{FALSE}$ and ⟨echo⟩ is not sent.
- SE corresponds to the case $\text{accept}_i = \text{FALSE}$ and ⟨echo⟩ been sent. Observe that once a process has sent ⟨echo⟩, its value of v_i does not interfere anymore with the subsequent control flow.
- AC corresponds to the case $\text{accept}_i = \text{TRUE}$ and ⟨echo⟩ been sent. A process only sets accept to TRUE if it has sent a message (or is about to do so in the current step).

Thus, the control state is captured within a single *status variable sv* over SV with the set $SV_0 = \{\text{V0}, \text{V1}\}$ of initial control states.

3.4 Specifications

Specifications are an encoding of the broadcast properties [34], which contain a safety property called *unforgeability*, and two liveness properties called *correctness* and *relay*:

$$\mathbf{G}\ ([\forall i.\ sv_i \neq \text{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \text{AC}]) \tag{U}$$

$$\mathbf{G}\ ([\forall i.\ sv_i = \text{V1}] \rightarrow \mathbf{F}\ [\exists j.\ sv_j = \text{AC}]) \tag{C}$$

$$\mathbf{G}\ ([\exists i.\ sv_i = \text{AC}] \rightarrow \mathbf{F}\ [\forall j.\ sv_j = \text{AC}]) \tag{R}$$

4 Experiments with SPIN

Figure 4 provides the central parts of the code of our case study. For the experiments we have implemented four distributed algorithms that use threshold-guarded commands, and differ in the fault model. We have one algorithm for each of the fault models discussed. In addition, the algorithms differ in the guarded commands. The following list is ordered from the most general fault model to the most restricted one. The given resilience conditions on n and t are the ones we expected from the literature, and their tightness was confirmed by our experiments:

Byz. tolerates t Byzantine faults if $n > 3t$,
symm. tolerates t symmetric (identical Byzantine [3]) faults if $n > 2t$,
omit. tolerates t send omission faults if $n > 2t$,
clean. tolerates t clean crash faults for $n > t$.

In addition, we verified a folklore reliable broadcasting algorithm that tolerates crash faults, which is given, e.g., in [9]. Further, we verified a Byzantine tolerant broadcasting algorithm from [6]. For the encoding of the algorithm from [6] we were required to use two message types — opposed to the one type of the ⟨echo⟩

Table 1. Summary of experiments related to [34]

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
				Byz				
B1	N=7,T=2,F=2	(U)	✓	3.13 sec.	74 MB	$193 \cdot 10^3$	$1 \cdot 10^6$	229
B2	N=7,T=2,F=2	(C)	✓	3.43 sec.	75 MB	$207 \cdot 10^3$	$2 \cdot 10^6$	229
B3	N=7,T=2,F=2	(R)	✓	6.3 sec.	77 MB	$290 \cdot 10^3$	$3 \cdot 10^6$	229
B4	N=7,T=3,F=2	(U)	✓	4.38 sec.	77 MB	$265 \cdot 10^3$	$2 \cdot 10^6$	233
B5	N=7,T=3,F=2	(C)	✓	4.5 sec.	77 MB	$271 \cdot 10^3$	$2 \cdot 10^6$	233
B6	N=7,T=3,F=2	(R)	✗	0.02 sec.	68 MB	$1 \cdot 10^3$	$13 \cdot 10^3$	210
				OMIT				
O1	N=5,To=2,Fo=2	(U)	✓	1.43 sec.	69 MB	$51 \cdot 10^3$	$878 \cdot 10^3$	175
O2	N=5,To=2,Fo=2	(C)	✓	1.64 sec.	69 MB	$60 \cdot 10^3$	$1 \cdot 10^6$	183
O3	N=5,To=2,Fo=2	(R)	✓	3.69 sec.	71 MB	$92 \cdot 10^3$	$2 \cdot 10^6$	183
O4	N=5,To=2,Fo=3	(U)	✓	1.39 sec.	69 MB	$51 \cdot 10^3$	$878 \cdot 10^3$	175
O5	N=5,To=2,Fo=3	(C)	✗	1.63 sec.	69 MB	$53 \cdot 10^3$	$1 \cdot 10^6$	183
O6	N=5,To=2,Fo=3	(R)	✗	0.01 sec.	68 MB	17	135	53
				SYMM				
S1	N=5,T=1,Fp=1,Fs=0	(U)	✓	0.04 sec.	68 MB	$3 \cdot 10^3$	$23 \cdot 10^3$	121
S2	N=5,T=1,Fp=1,Fs=0	(C)	✓	0.03 sec.	68 MB	$3 \cdot 10^3$	$24 \cdot 10^3$	121
S3	N=5,T=1,Fp=1,Fs=0	(R)	✓	0.08 sec.	68 MB	$5 \cdot 10^3$	$53 \cdot 10^3$	121
S4	N=5,T=3,Fp=3,Fs=1	(U)	✓	0.01 sec.	68 MB	66	267	62
S5	N=5,T=3,Fp=3,Fs=1	(C)	✗	0.01 sec.	68 MB	62	221	66
S6	N=5,T=3,Fp=3,Fs=1	(R)	✓	0.01 sec.	68 MB	62	235	62
				CLEAN				
C1	N=3,Tc=2,Fc=2,Fnc=0	(U)	✓	0.01 sec.	68 MB	668	$7 \cdot 10^3$	77
C2	N=3,Tc=2,Fc=2,Fnc=0	(C)	✓	0.01 sec.	68 MB	892	$8 \cdot 10^3$	81
C3	N=3,Tc=2,Fc=2,Fnc=0	(R)	✓	0.02 sec.	68 MB	$1 \cdot 10^3$	$17 \cdot 10^3$	81

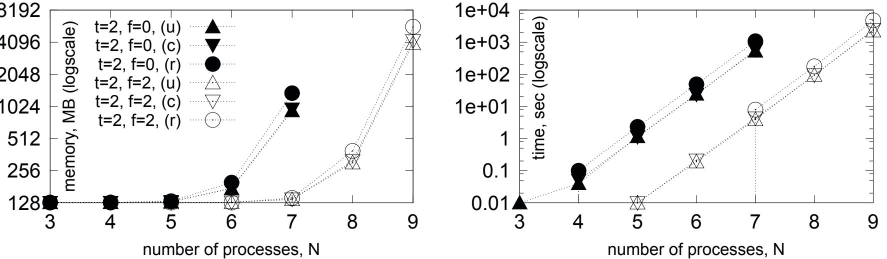


Fig. 5. Spin memory usage (left) and running time (right) for Byz

messages in Algorithm 1. Finally, we implemented the asynchronous condition-based consensus algorithm from [27]. We specialized it to binary consensus, which resulted in an encoding which requires four different message types.

The major goal of the experiments was to check the adequacy of our formalization. To this end, we first considered the four well-understood variants of [34], for each of which we systematically changed the parameter values. By doing so,

Table 2. Summary of experiments with algorithms from [9,6,27]

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
			Folklore Broadcast [9]					
F1	N=2	(U)	✓	0.01 sec.	98 MB	121	$7 \cdot 10^3$	77
F2	N=2	(R)	✓	0.01 sec.	98 MB	143	$8 \cdot 10^3$	48
F3	N=2	(F)	✓	0.01 sec.	98 MB	257	$2 \cdot 10^3$	76
F4	N=6	(U)	✓	386 sec.	670 MB	$15 \cdot 10^6$	$20 \cdot 10^6$	272
F5	N=6	(R)	✓	691 sec.	996 MB	$24 \cdot 10^6$	$370 \cdot 10^6$	272
F6	N=6	(F)	✓	1690 sec.	1819 MB	$39 \cdot 10^6$	$875 \cdot 10^6$	328
			Asynchronous Byzantine Agreement [6]					
T1	N=5,T=1,F=1	(R)	✓	131 sec.	239 MB	$4 \cdot 10^6$	$74 \cdot 10^6$	211
T2	N=5,T=1,F=2	(R)	✗	0.68 sec.	99 MB	$11 \cdot 10^3$	$465 \cdot 10^3$	187
T3	N=5,T=2,F=2	(R)	✗	0.02 sec.	99 MB	726	$9 \cdot 10^3$	264
			Condition-based consensus [27]					
S1	N=3,T=1,F=1	(V0)	✓	0.01 sec.	98 MB	$1.4 \cdot 10^3$	$7 \cdot 10^3$	115
S2	N=3,T=1,F=1	(V1)	✓	0.04 sec.	98 MB	$3 \cdot 10^3$	$18 \cdot 10^3$	128
S3	N=3,T=1,F=1	(A)	✓	0.09 sec.	98 MB	$8 \cdot 10^3$	$42 \cdot 10^3$	127
S4	N=3,T=1,F=1	(T)	✓	0.16 sec.	66 MB	$9 \cdot 10^3$	$83 \cdot 10^3$	133
S5	N=3,T=1,F=2	(V0)	✓	0.02 sec.	68 MB	1724	9835	123
S6	N=3,T=1,F=2	(V1)	✓	0.05 sec.	68 MB	3647	$23 \cdot 10^3$	136
S7	N=3,T=1,F=2	(A)	✓	0.12 sec.	68 MB	$10 \cdot 10^3$	$55 \cdot 10^3$	135
S8	N=3,T=1,F=2	(T)	✗	0.05 sec.	68 MB	$3 \cdot 10^3$	$17 \cdot 10^3$	135

we verify that under our modeling the different combination of parameters lead to the expected result. Table 1 and Figure 5 summarize the results of our experiments for broadcasting algorithms in the spirit of [34]. Lines B1 – B3, O1 – O3, S1 – S3, and C1 – C3 capture the cases that are within the resilience condition known for the respective algorithm, and the algorithms were verified by Spin. In Lines B4 – B6, the algorithm's parameters are chosen to achieve a goal that is known to be impossible [28], i.e., to tolerate that 3 out of 7 processes may fail. This violates the $n > 3t$ requirement. Our experiment shows that even if only 2 faults occur in this setting, the relay specification (R) is violated. In Lines O4 – O6, the algorithm is designed properly, i.e., 2 out of 5 processes may fail ($n > 2t$ in the case of omission faults). Our experiments show that this algorithm fails in the presence of 3 faulty processes, i.e., (C) and (R) are violated.

Table 2 summarizes our experiments for the algorithms in [9], [6], and [27]. The specification (F) is related to agreement and was also used in [17]. Properties (V0) and (V1) are non-triviality, that is, if all processes propose 0 (1), then 0 (1) is the only possible decision value. Property (A) is agreement and similar to (R), while Property (T) is termination, and requires that every correct process eventually decides. In all experiments the validity of the specifications was as expected from the distributed algorithms literature.

For slightly bigger systems, that is, for $n = 11$ our experiments run out of memory. This shows the need for parameterized verification of these algorithms.

5 Related Work

As fault tolerance is required to increase the reliability of systems, the verification of fault tolerance mechanisms is an important challenge. There are two classes of approaches towards fault tolerance, namely *fault detection*, and *fault masking.*

Methods in the first class follow the fault detection, isolation, and recovery (FDIR) principles: at runtime one tries to detect faults and to automatically perform counter measures. In this area, in [32] Spin was used to validate a design based on the well-known primary backup idea. Under the FDIR approach, validation techniques have also been introduced in [15,8,19].

However, it is well understood that it is not always possible to reliably detect faults; for instance, in asynchronous distributed systems it is not possible to distinguish a process that prematurely stopped from a slow process, and in synchronous systems there are cases where the border between correct and faulty behavior cannot be drawn sharply [1]. To address such issues, fault masking has been introduced. Here, one does not try to detect or isolate faults, but tries to keep those components operating consistently that are not directly hit by faults, cf. distributed agreement [28]. The fault-tolerant distributed algorithms that we consider in this paper belong to this approach.

Specific masking fault-tolerant distributed algorithms have been verified, e.g., a consensus algorithm in [36], and a clock synchronization algorithm in [35]. In [25], a bug has been found in a previously published clock synchronization algorithm that was supposed to tolerate Byzantine faults.

Formalization and verification of a class of fault-tolerant distributed algorithms have been addressed in [5]. Their formalization uses the fact that for many distributed algorithms it is relevant how many messages are received, but the order in which they are received is not important. They provide a framework for such algorithms and show that these algorithms can be efficiently verified using partial order reduction. While in this work we consider similar message counting ideas, our formalization targets at parameterized model checking [21] rather than partial order reductions for systems of small size.

6 Conclusions

In this paper we presented a way to efficiently encode fault-tolerant threshold-guarded distributed algorithms using shared variables. We showed that our encoding scales significantly better than a straightforward approach. With this encoding we were able to verify small system instances of a number of broadcasting algorithms [34,6,9] for diverse failure models. We could also find counter examples in cases where we knew from theory that the given number of faults cannot be tolerated. We also verified a condition-based consensus algorithm [27].

As our mid-term goal is to verify state-of-the-art fault-tolerant distributed algorithms, there are several follow-up steps we are taking. In [21] we show that the encoding we described in this paper is a basis for parameterized model checking techniques that allow us to verify distributed algorithms for any system

size. We have already verified some of the algorithms mentioned above, while we are still working on techniques to verify the others. Also we are currently working on verification of the Paxos-like Byzantine consensus algorithm from [26], which is also threshold-guarded. The challenges of this algorithm are threefold. First, it consists of three different process types — proposers, accepters, learners — while the algorithms discussed in this paper are just compositions of processes of the same type. Second, to tolerate a single fault, the algorithm requires at least four proposers, six acceptors, and four learners. Our preliminary experiments show that 14 processes is a challenge for explicit state enumeration. Third, as the algorithm solves consensus, it cannot work in the asynchronous model [16], and we have to restrict the interleavings of steps, and the message delays.

References

1. Ademaj, A.: Slightly-off-specification failures in the time-triggered architecture. In: High-Level Design Validation and Test Workshop, pp. 7–12. IEEE (2002)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with Byzantine failures and little system synchrony. In: DSN, pp. 147–155 (2006)
3. Attiya, H., Welch, J.: Distributed Computing, 2nd edn. John Wiley & Sons (2004)
4. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theoretical Computer Science 412(40), 5602–5630 (2011)
5. Bokor, P., Kinder, J., Serafini, M., Suri, N.: Efficient model checking of fault-tolerant distributed protocols. In: DSN, pp. 73–84 (2011)
6. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32(4), 824–840 (1985)
7. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. 81, 13–31 (1989)
8. Bucchiarone, A., Muccini, H., Pelliccione, P.: Architecting fault-tolerant component-based systems: from requirements to testing. Electr. Notes Theor. Comput. Sci. 168, 77–90 (2007)
9. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
10. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
11. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
12. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
13. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
14. Emerson, E., Namjoshi, K.: Reasoning about rings. In: POPL, pp. 85–94 (1995)
15. Feather, M.S., Fickas, S., Razermera-Mamy, N.A.: Model-checking for validation of a fault protection system. In: HASE, pp. 32–41 (2001)
16. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)

17. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 315–331. Springer, Heidelberg (2008)
18. Függer, M., Schmid, U.: Reconciling fault-tolerant distributed computing and systems-on-chip. Distributed Computing 24(6), 323–355 (2012)
19. Gnesi, S., Latella, D., Lenzini, G., Abbaneo, C., Amendola, A., Marmo, P.: A formal specification and validation of a critical system in presence of Byzantine errors. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 535–549. Springer, Heidelberg (2000)
20. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
21. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Brief announcement: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: ACM PODC (to appear, 2013) (long version at arXiv CoRR abs/1210.3846)
22. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
23. Lamport, L.: On interprocess communication. Part I: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
24. Lynch, N.: Distributed Algorithms. Morgan Kaufman, San Francisco (1996)
25. Malekpour, M.R., Siminiceanu, R.: Comments on the "Byzantine self-stabilizing pulse synchronization". protocol: Counterexamples. Tech. rep., NASA (February 2006)
26. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Trans. Dep. Sec. Comp. 3(3), 202–215 (2006)
27. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN, pp. 541–550 (2003)
28. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)
29. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0,1,\infty)$- counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
30. Powell, D.: Failure mode assumptions and assumption coverage. In: FTCS-22, Boston, MA, USA, pp. 386–395 (1992)
31. Santoro, N., Widmayer, P.: Time is not a healer. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–313. Springer, Heidelberg (1989)
32. Schneider, F., Easterbrook, S.M., Callahan, J.R., Holzmann, G.J.: Validating requirements for fault tolerant systems using model checking. In: ICRE, pp. 4–13 (1998)
33. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)
34. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing 2, 80–94 (1987)
35. Steiner, W., Rushby, J.M., Sorea, M., Pfeifer, H.: Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In: DSN, pp. 189–198 (2004)
36. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distributed Computing 23(5-6), 341–358 (2011)
37. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distributed Computing 20(2), 115–140 (2007)

Chapter 2

Accuracy of Message Counting Abstraction in Fault-Tolerant Distributed Algorithms

Igor Konnov, Josef Widder, Francesco Spegni, and Luca Spalazzi. Accuracy of Message Counting Abstraction in Fault-Tolerant Distributed Algorithms. VMCAI, vol. 10145 of *LNCS*, pp. 347–366, 2017.

DOI: http://dx.doi.org/10.1007/978-3-319-52234-0_19

Accuracy of Message Counting Abstraction in Fault-Tolerant Distributed Algorithms

Igor Konnov[1( )], Josef Widder[1], Francesco Spegni[2], and Luca Spalazzi[2]

[1] TU Wien (Vienna University of Technology), Vienna, Austria
konnov@forsyte.at
[2] UnivPM, Ancona, Italy

Abstract. Fault-tolerant distributed algorithms are a vital part of mission-critical distributed systems. In principle, automatic verification can be used to ensure the absence of bugs in such algorithms. In practice however, model checking tools will only establish the correctness of distributed algorithms if message passing is encoded efficiently. In this paper, we consider abstractions suitable for many fault-tolerant distributed algorithms that count messages for comparison against thresholds, e.g., the size of a majority of processes. Our experience shows that storing only the numbers of sent and received messages in the global state is more efficient than explicitly modeling message buffers or sets of messages. Storing only the numbers is called message-counting abstraction. Intuitively, this abstraction should maintain all necessary information. In this paper, we confirm this intuition for asynchronous systems by showing that the abstract system is bisimilar to the concrete system. Surprisingly, if there are real-time constraints on message delivery (as assumed in fault-tolerant clock synchronization algorithms), then there exist neither timed bisimulation, nor time-abstracting bisimulation. Still, we prove this abstraction useful for model checking: it preserves ATCTL properties, as the abstract and the concrete models simulate each other.

1 Introduction

The following algorithmic idea is pervasive in fault-tolerant distributed computing [13,21,30,33,36,39]: each correct process counts messages received from distinct peers. Then, given the total number of processes n and the maximum number of faulty processes t, a process performs certain actions only if the message counter reaches a threshold such as $n - t$ (this number ensures that faulty processes alone cannot prevent progress in the computation). A list of benchmark algorithms that use such thresholds can be found in [27]. On the left of Fig. 1, we give an example pseudo code [36]. This algorithm works in a timed environment [35] (with a time bound τ^+ on message delays) in the presence of Byzantine faults ($n > 3t$) and provides safety and liveness guarantees such as:

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

© Springer International Publishing AG 2017
A. Bouajjani and D. Monniaux (Eds.): VMCAI 2017, LNCS 10145, pp. 347–366, 2017.
DOI: 10.1007/978-3-319-52234-0_19

```
1  local myval_i ∈ {0,1}                          21 local myval_i ∈ {0,1}
2                                                 22 global nsntEcho ∈ N_0 initially 0
3                                                 23 local hasSent ∈ B initially F
4                                                 24 local rcvdEcho ∈ N_0 initially 0
5                                                 25
6  do atomically                                  26 do atomically
7   -- messages are received implicitly           27  if (*) -- choose non-deterministically
8   if myval_i = 1                                28   and rcvdEcho < nsntEcho + f
9    and not sent ECHO before                     29  then rcvdEcho++;
10  then send ECHO to all                         30
11                                                31  if myval_i = 1 and hasSent = F
12  if received ECHO                              32  then { nsntEcho++; hasSent = T; }
13      from at least t + 1 distinct processes    33
14   and not sent ECHO before                     34
15  then send ECHO to all                         35  if rcvdEcho ≥ t + 1 and hasSent = F
16                                                36  then { nsntEcho++; hasSent = T; }
17  if received ECHO                              37
18      from at least n − t distinct processes    38  if rcvdEcho ≥ n − t
19  then accept                                   39  then accept
20 od                                             40 od
```

Fig. 1. Pseudocode of a broadcast primitive to simulate authenticated broadcast [36] (left), and pseudocode of its message-counting abstraction (right)

(a) If a correct process accepts (that is, executes Line 19) at time T, then all correct processes accept by time $T + 2\tau^+$.
(b) If all correct processes start with $myval_i = 0$, then no correct process ever *accepts*.
(c) If all correct processes start with $myval_i = 1$, then at least one correct process eventually accepts.

As is typical for the distributed algorithms literature, the pseudo code from Fig. 1 omits "unnecessary book-keeping" details of message passing. That is, neither the local data structures that store the received messages nor the message buffers are explicitly described. Hence, if we want to automatically verify such an algorithm design, it is up to a verification expert to find adequate modeling and proper abstractions of message passing.

The authors of [23] suggested to model message passing using message counters instead of keeping track of individual messages. This modeling was shown experimentally to be efficient for fixed size systems, and later a series of parameterized model checking techniques was based upon it [22,23,25–27]. The encoding on the right of Fig. 1 is obtained by adding a global integer variable `nsntEcho`. Incrementing this variable (Line 36) encodes that a correct process executes Line 15 of the original pseudo code. The ith process keeps the number of received messages in a local integer variable $\texttt{rcvdEcho}_i$ that can be increased, as long as the invariant $\texttt{rcvdEcho}_i \leq \texttt{nsntEcho} + f$ is preserved, where f is the actual number of Byzantine faulty processes in the run. (This models that correct processes can receive up to f messages sent by faulty processes.) In fact, this modeling can be seen as a *message-counting abstraction* of a distributed system that uses message buffers.

The broadcast primitive in Fig. 1 is also used in the seminal clock synchronization algorithm from [35]. For clock synchronization, the precision of the

clocks depends on the timing behavior[1] of the message system that the processes use to re-synchronize; e.g., in [35] it is required that each message sent at an instant T by a correct process must be delivered by a correct recipient process in the time interval $[T + \tau^-, T + \tau^+]$ for some bounds τ^- and τ^+ fixed in each run.

The standard theory of timed automata [7] does not account for message passing directly. To incorporate messages, one specifies a message passing system as a network of timed automata, i.e., a collection of timed automata that are scheduled with respect to interleaving semantics and interact via rendezvous, synchronous broadcast, or shared variables [12]. In this case, there are two typical ways to encode message passing: (i) for each pair of processes, introduce a separate timed automaton that models a channel between the processes, or (ii) introduce a single timed automaton that stores messages from timed automata (modeling the processes) and delivers the messages later by respecting the timing constraints. The same applies to Timed I/O automata [24]. Both solutions maintain much more details than required for automated verification of distributed algorithms such as [35]: First, processes do not compare process identifiers when making transitions, and thus are symmetric. Second, processes do not compare identifiers in the received messages, but only count messages.

For automated verification purposes, it appears natural to model such algorithms with timed automata that use a message-counting abstraction. However, the central question for practical verification is: *how precise is the message-counting abstraction?* In other words, given an algorithm A, what is the strongest equivalence between the model $M_S(A)$ using message sets and the model $M_C(A)$ using message counting. If the message counting abstraction is too coarse, then this may lead to spurious counterexamples, which may result in many refinement steps [17], or even may make the verification procedure incomplete.

Contributions. We introduce timed and untimed models suitable for the verification of threshold-based distributed algorithms, and establish relations between these models. An overview of the following contributions is depicted in Fig. 2:

- We define a model of processes that count messages. We then compose them into asynchronous systems (interleaving semantics). We give two variants: message passing, where the messages are stored in sets, and message counting, where only the number of sent messages is stored in shared variables.
- We then show that in the asynchronous case, the message passing and the message counting variants are bisimilar. This proves the intuition that underlies the verification results from [22,23,25,27]. It explains why no spurious counterexamples due to message-counting abstraction were experienced in the experimental evaluation of the verification techniques from [22].

[1] As we deal with distributed algorithms and timed automata, the notion of a *clock* appears in two different contexts in this paper, which should not be confused: The problem of clock synchronization is to compute adjustment for the hardware clocks (oscillators). In the context of timed automata, clocks are special variables used to model the timing behavior of a system.

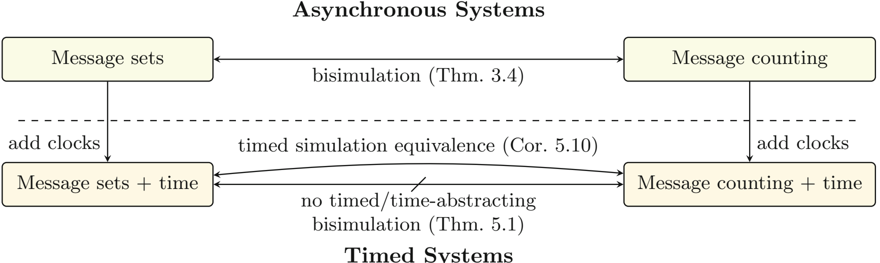


Fig. 2. Relationship between different modeling choices.

- We obtain timed models by adding timing constraints on message delays that restrict the message reception time depending on the sending times.
- We prove the surprising result that, in general, there is neither timed bisimulation nor time-abstracting bisimulation between the message passing and the message counting variants.
- Finally, we prove that there is timed simulation equivalence between the message passing and the message counting variants. This paves a way for abstraction-based model checking of timed distributed algorithms [35].

In the following section, we briefly recall the classic definitions of transition systems, timed automata, and simulations [7,16]. However, the timed automata defined there do not provide standard means to express processes that communicate via asynchronous message passing, as required for distributed algorithms. As we are interested in timed automata that capture this structure, we first define asynchronous message passing in Sect. 3 and then add timing constraints in Sect. 4 via message sets and message counting.

2 Preliminaries

We recall the classic definitions to the extent necessary for our work, and add two non-standard notions: First, our definition of a timed automaton assumes partitioning of the set of clocks into two disjoint sets: the message clocks (used to express the timing constraints of the message system underlying the distributed algorithm) and the specification clocks (used to express the specifications). Second, we assume that clocks are "not ticking" before they are started (more precisely, they are initialized to $-\infty$).

We will use the following sets: the set of Boolean values $\mathbb{B} = \{\mathsf{F}, \mathsf{T}\}$, the set of natural numbers $\mathbb{N} = \{1, 2, \dots\}$, the set $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, the set of non-negative reals $\mathbb{R}_{\geq 0}$, and the set of time instants $\mathbb{T} := \mathbb{R}_{\geq 0} \cup \{-\infty\}$.

Transition Systems. Given a finite set AP of atomic propositions, a *transition system* is a tuple $TS = (S, S^0, R, L)$ where S is a set of states, $S^0 \subseteq S$ are the initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{\mathsf{AP}}$ is a labeling function.

Clocks. A clock is a variable that ranges over the set $\mathbb{T}$. We call a clock that has the value $-\infty$ *uninitialized*. For a set X of clocks, a *clock valuation* is a function $\nu : X \to \mathbb{T}$. Given a clock valuation ν and a $\delta \in \mathbb{R}_{\geq 0}$, we define $\nu + \delta$ to be the valuation ν' such that $\nu'(c) = \nu(c) + \delta$ for $c \in X$ (Note that $-\infty + \delta = -\infty$). For a set $Y \subseteq X$ and a clock valuation $\nu : X \to \mathbb{T}$, we denote by $\nu[Y := 0]$ the valuation ν' such that $\nu'(c) = 0$ for $c \in Y \cap X$ and $\nu'(c) = \nu(c)$ for $c \in X \setminus Y$. Given a set of clocks Z, the set of *clock constraints* $\Psi(Z)$ is defined to contain all expressions generated by the following grammar:

$$\zeta := c \leq a \mid c \geq a \mid c < a \mid c > a \mid \zeta \wedge \zeta \qquad \text{for } c \in Z, a \in \mathbb{N}_0$$

Timed Automata. Given a set of atomic propositions AP and a finite transition system (S, S^0, R, L) over AP, which models discrete control of a system, we model the system's real-time behavior with a *timed automaton*, i.e., a tuple $TA = (S, S^0, R, L, X \cup U, I, E)$ with the following properties:

- The set $X \cup U$ is the disjoint union of the sets of *message clocks* X and *specification clocks* U.
- The function $I : S \to \Psi(X \cup U)$ is a *state invariant*, which assigns to each discrete state a clock constraint over $X \cup U$, which must hold in that state. We denote by $\mu, \nu \models I(s)$ that the clock valuations μ and ν satisfy the constraints of $I(s)$.
- $E : R \to \Psi(X \cup U) \times 2^{(X \cup U)}$ is a *state switch relation* that assigns to each transition a guard on clock values and a (possibly empty) set of clocks that must be reset to zero, when the transition takes place.

We assume that AP is disjoint from $\Psi(X \cup U)$. Thus, the discrete behavior does not interfere with propositions on time. The semantics of a timed automaton $TA = (S, S^0, R, L, X \cup U, I, E)$ is an infinite transition system $TS(TA) = (Q, Q^0, \Delta, \lambda)$ over propositions $\mathsf{AP} \cup \Psi(U)$ with the following properties [6]:

1. The set Q of states consists of triples (s, μ, ν), where $s \in S$ is the discrete component of the state, whereas $\mu : X \to \mathbb{T}$ and $\nu : U \to \mathbb{T}$ are valuations of the message and specification clocks respectively such that $\mu, \nu \models I(s)$.
2. The set $Q^0 \subseteq Q$ of initial states comprises triples (s_0, μ_0, ν_0) with $s_0 \in S_0$, and clocks are set to $-\infty$, i.e., $\forall c \in X.\ \mu_0(c) = -\infty$ and $\forall c \in U.\ \nu_0(c) = -\infty$.
3. The transition relation Δ contains pairs $((s, \mu, \nu), (s', \mu', \nu'))$ of two kinds of transitions:
   (a) A time step: $s' = s$ and $\mu' = \mu + \delta$, $\nu' = \nu + \delta$, for $\delta > 0$, provided that for all $\delta' :\ 0 \leq \delta' \leq \delta$ the invariant is preserved, i.e., $\mu + \delta', \nu + \delta' \models I(s)$.
   (b) A discrete step: there is a transition $(s, s') \in R$ with $(\varphi, Y) = E((s, s'))$ whose guard φ is enabled, i.e., $\mu, \nu \models \varphi$, and the clocks from Y are reset, i.e., $\mu' = \mu[Y \cap X := 0]$, $\nu' = \nu[Y \cap U := 0]$, provided that $\mu', \nu' \models I(s)$.

   Given a transition $(q, q') \in \Delta$, we write $q \xrightarrow{\delta}_\Delta q'$ for a time step with delay $\delta \in \mathbb{R}_{\geq 0}$, or $q \to_\Delta q'$ for a discrete step.
4. The labeling function $\lambda : Q \to 2^{\mathsf{AP} \cup \Psi(U)}$ is defined as follows. For any state $q = (s, \mu, \nu)$, the labeling $\lambda(q) = L(s) \cup \{\varphi \in \Psi(U) : \mu, \nu \models \varphi\}$.

Comparing System Behaviors. For transition systems $TS_i = (S_i, S_i^0, R_i, L_i)$ for $i \in \{1, 2\}$, a relation $H \subseteq S_1 \times S_2$ is a *simulation*, if (i) for each $(s_1, s_2) \in H$ the labels coincide $L_1(s_1) = L_2(s_2)$, and (ii) for each transition $(s_1, t_1) \in R_1$, there is a transition $(s_2, t_2) \in R_2$ such that $(t_1, t_2) \in H$. If, in addition, the set $H^{-1} = \{(s_2, s_1) \colon (s_1, s_2) \in H\}$ is also a simulation, then H is called *bisimulation*.

Further, if TA_1 and TA_2 are timed automata with $TS(TA_i) = (Q_i, Q_i^0, \Delta_i, \lambda_i)$ for $i \in \{1, 2\}$, then a simulation $H \subseteq Q_1 \times Q_2$ is called *timed simulation* [29], and a bisimulation $B \subseteq Q_1 \times Q_2$ is called *timed bisimulation* [15].

For transition systems $TS_i = (S_i, S_i^0, R_i, L_i)$ for $i \in \{1, 2\}$, we say that a simulation $H \subseteq S_1 \times S_2$ is *initial*, if $\forall s \in S_1^0\ \exists t \in S_2^0.\ (s, t) \in H$. A bisimulation $B \subseteq S_1 \times S_2$ is initial, if the simulations B and B^{-1} are initial. The same applies to timed (bi-)simulations. Then, for $i \in \{1, 2\}$, we recall the standard preorders and equivalences on a pair of transition systems $TS_i = (S_i, S_i^0, R_i, L_i)$, and on a pair of timed automata TA_i, where $TS(TA_i) = (Q_i, Q_i^0, \Delta_i, \lambda_i)$:

1. $TS_1 \approx TS_2$ (*bisimilar*), if there is an initial bisimulation $B \subseteq S_1 \times S_2$.
2. $TA_1 \preceq^t TA_2$ (TA_2 *time-simulates* TA_1), if there is an initial timed simulation $H \subseteq Q_1 \times Q_2$.
3. $TA_1 \approx^t TA_2$ (*time-bisimilar*), if there is an initial timed bisimulation $B \subseteq Q_1 \times Q_2$.
4. $TA_1 \simeq^t TA_2$ (*time-simulation equivalent*), if $TA_1 \preceq^t TA_2$ and $TA_2 \preceq^t TA_1$.

Timed bisimulation forces time steps to advance clocks by the same amount of time. A coarser relation — called time-abstracting bisimulation [37] — allows two transition systems to advance clocks at "different speeds". Given two timed automata TA_i, for $i \in \{1, 2\}$ and the respective transition systems $TS(TA_i) = (Q_i, Q_i^0, \Delta_i, \lambda_i)$, a binary relation $B \subseteq Q_1 \times Q_2$ is a *time-abstracting bisimulation* [37], if the following holds for every pair $(q_1, q_2) \in B$:

1. The labels coincide: $\lambda_1(q_1) = \lambda_2(q_2)$;
2. For all j and k such that $\{j, k\} = \{1, 2\}$, and each discrete step $q_j \rightarrow_{\Delta_j} r_j$, there is a discrete step $q_k \rightarrow_{\Delta_k} r_k$ and $(r_j, r_k) \in B$;
3. For all j and k such that $\{j, k\} = \{1, 2\}$, a delay $\delta \in \mathbb{R}_{\geq 0}$, and a time step $q_j \xrightarrow{\delta}_{\Delta_j} r_j$, there is a delay $\delta' \in \mathbb{R}_{\geq 0}$ and a time step $q_k \xrightarrow{\delta'}_{\Delta_k} r_k$ such that $(r_j, r_k) \in B$.

By substituting δ' with δ, one obtains the definition of timed bisimulation.

3 Asynchronous Message Passing Systems

Timed automata as defined above neither capture processes nor communication via messages, as would be required to model distributed algorithms. Hence we now introduce these notions and then construct an asynchronous system using processes and message passing (or message counting). We assume that at every step a process receives and sends at most one message [19]. In Sect. 4, we add time to this modeling in order to obtain a timed automaton.

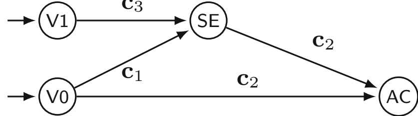


Fig. 3. A graphical representation of a process discussed in Example 3.1

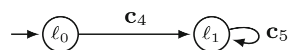


Fig. 4. A simple two-state process (used later for Theorem 5.1)

Single Correct Process. We assume a (possibly infinite) set of control states $\mathcal{L}$ and a subset $\mathcal{L}_0 \subseteq \mathcal{L}$ of initial control states. We fix a finite set MT of message types. We assume that the control states in $\mathcal{L}$ keep track of the messages sent by a process. Thus, $\mathcal{L}$ comes with a predicate $\mathsf{is_sent}\colon \mathcal{L} \times \mathsf{MT} \to \mathbb{B}$, where $\mathsf{is_sent}(\ell, m)$ evaluates to true if and only if a message of type m has been sent according to the control state ℓ. Finally, we introduce a set Π of parameters and store the parameter values in a vector $\mathbf{p} \in {\mathbb{N}_0}^{|\Pi|}$. As noted in [22], parameter values are typically restricted with a resilience condition such as $n > 3t$ (less than a third of the processes are faulty), so we will assume that there is a set of all admissible combinations of parameter values $\mathbf{P}_{RC} \subseteq {\mathbb{N}_0}^{|\Pi|}$.

The behavior of a single process is defined as a *process transition relation* $\mathcal{T} \subseteq \mathcal{L} \times {\mathbb{N}_0}^{|\Pi|} \times {\mathbb{N}_0}^{|\mathsf{MT}|} \times \mathcal{L}$ encoding transitions guarded by conditions on message counters that range over ${\mathbb{N}_0}^{|\mathsf{MT}|}$: when $(\ell, \mathbf{p}, \mathbf{c}, \ell') \in \mathcal{T}$, a process can make a transition from the control state ℓ to the control state ℓ', provided that, for every $m \in \mathsf{MT}$, the number of received messages of type m is greater than or equal to $\mathbf{c}(m)$ in a configuration with parameter values $\mathbf{p}$.

Example 3.1. The process shown in Fig. 1 can be written in our definitions as follows. The algorithm is using only one message type, and thus $\mathsf{MT} = \{\mathsf{ECHO}\}$. We assume a set of control states $\mathcal{L} = \{\mathsf{V0}, \mathsf{V1}, \mathsf{SE}, \mathsf{AC}\}$: $\mathsf{V0}$ and $\mathsf{V1}$ encode the initial states where $myval = 0$ and $myval = 1$ respectively, $\mathsf{pc} = \mathsf{SE}$ encodes the status "ECHO sent before", and $\mathsf{pc} = \mathsf{AC}$ encodes the status "accept". The initial control states are: $\mathcal{L}_0 = \{\mathsf{V0}, \mathsf{V1}\}$. The transition relation contains four types of transitions: $t_1^{\mathbf{p}} = (\mathsf{V0}, \mathbf{p}, \mathbf{c}_1, \mathsf{SE})$, $t_2^{\mathbf{p}} = (\mathsf{V0}, \mathbf{p}, \mathbf{c}_2, \mathsf{AC})$, $t_3^{\mathbf{p}} = (\mathsf{V1}, \mathbf{p}, \mathbf{c}_3, \mathsf{SE})$, and $t_4^{\mathbf{p}} = (\mathsf{SE}, \mathbf{p}, \mathbf{c}_2, \mathsf{AC})$, for any $\mathbf{p} \in {\mathbb{N}_0}^{|\Pi|}$ and $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$ satisfying the following: $\mathbf{c}_1(\mathsf{ECHO}) \geq \mathbf{p}(t) + 1$, $\mathbf{c}_2(\mathsf{ECHO}) \geq \mathbf{p}(n) - \mathbf{p}(t)$, and $\mathbf{c}_3(\mathsf{ECHO}) \geq 0$. Finally, $\mathsf{is_sent}(\ell, \mathsf{ECHO})$ iff $\ell \in \{\mathsf{SE}, \mathsf{AC}\}$. A concise graphical representation of the transition relation is given in Fig. 3. There, each edge represents multiple transitions of the same type. Let us observe that while the action of sending a message can be inferred by simply checking all the transitions going from a state s to a state t such that $\neg\mathsf{is_sent}(s)$ and $\mathsf{is_sent}(t)$, the action of receiving an individual message is not part of the process description at this level. However, if a guarded transition is taken, this implies that a threshold has been reached, e.g., in case of $\mathbf{c}_1$, at least $t + 1$ messages were received. $\triangleleft$

Table 1. The message-passing and message-counting interpretations

Message passing (MP)	Message counting (MC):
$Msg_{\mathsf{MP}} \triangleq \mathsf{MT} \times \mathsf{Proc}$	$Msg_{\mathsf{MC}} \triangleq \mathsf{MT} \times \{\mathsf{C}, \mathsf{F}\}$
$MsgSets_{\mathsf{MP}} \triangleq 2^{\mathsf{MT} \times \mathsf{Proc}}$	$MsgSets_{\mathsf{MC}} \triangleq \{0, \dots, \lvert\mathsf{Corr}\rvert\}^{\lvert\mathsf{MT}\rvert} \times \{0, \dots, \lvert\mathsf{Byz}\rvert\}^{\lvert\mathsf{MT}\rvert}$
Initial messages, $init \in MsgSets$	
$init_{\mathsf{MP}} \triangleq \emptyset$	$init_{\mathsf{MC}} \triangleq ((0, \dots, 0), (0, \dots, 0))$
Count messages, $card : \mathsf{MT} \times MsgSets \to \mathbb{N}_0$	
$card_{\mathsf{MP}}(m, M) \triangleq \lvert\{p \in \mathsf{Proc} \colon (m, p) \in M\}\rvert$	$card_{\mathsf{MC}}(m, (\mathbf{c}_{\mathsf{C}}, \mathbf{c}_{\mathsf{F}})) \triangleq \mathbf{c}_{\mathsf{C}}(m) + \mathbf{c}_{\mathsf{F}}(m)$
Add a message, $add : Msg \times MsgSets \to MsgSets$	
$add_{\mathsf{MP}}(\langle m, p\rangle, M) \triangleq M \cup \{\langle m, p\rangle\}$	$add_{\mathsf{MC}}((m, tag), (\mathbf{c}_{\mathsf{C}}, \mathbf{c}_{\mathsf{F}})) \triangleq (\mathbf{c}'_{\mathsf{C}}, \mathbf{c}'_{\mathsf{F}})$ such that $\mathbf{c}'_{\mathsf{C}}(m) = \mathbf{c}_{\mathsf{C}}(m) + 1$ and $\mathbf{c}'_{\mathsf{F}}(m) = \mathbf{c}_{\mathsf{F}}(m)$, if $tag = \mathsf{C}$ $\mathbf{c}'_{\mathsf{F}}(m) = \mathbf{c}_{\mathsf{F}}(m) + 1$ and $\mathbf{c}'_{\mathsf{C}}(m) = \mathbf{c}_{\mathsf{C}}(m)$, if $tag = \mathsf{F}$ and $\mathbf{c}'(m') = \mathbf{c}(m)$ for $m' \in \mathsf{MT}, m' \neq m$
Is there a message to deliver? $inTransit : Msg \times MsgSets \times MsgSets \to \mathbb{B}$	
$inTransit_{\mathsf{MP}}(\langle m, p\rangle, M, M') \triangleq$ $(p \in \mathsf{Corr} \wedge \langle m, p\rangle \in M' \setminus M) \vee (p \in \mathsf{Byz} \wedge \langle m, p\rangle \notin M)$	$inTransit_{\mathsf{MC}}((m, tag), (\mathbf{c}_{\mathsf{C}}, \mathbf{c}_{\mathsf{F}}), (\mathbf{c}'_{\mathsf{C}}, \mathbf{c}'_{\mathsf{F}})) \triangleq$ $(tag = \mathsf{C} \wedge \mathbf{c}'_{\mathsf{C}} > \mathbf{c}_{\mathsf{C}}) \vee (tag = \mathsf{F} \wedge \mathbf{c}_{\mathsf{F}} < \lvert\mathsf{Byz}\rvert)$

We make two assumptions typical for distributed algorithms [19,35]:

A1 Processes do not forget that they have sent messages: If $(\ell, \mathbf{p}, \mathbf{c}, \ell') \in \mathcal{T}$, then $\mathsf{is_sent}(\ell, m) \to \mathsf{is_sent}(\ell', m)$ for every $m \in \mathsf{MT}$.
A2 At each step a process sends at most one message to all: If $(\ell, \mathbf{p}, \mathbf{c}, \ell') \in \mathcal{T}$ and $\neg\mathsf{is_sent}(\ell, m) \wedge \mathsf{is_sent}(\ell', m) \wedge \neg\mathsf{is_sent}(\ell, m') \wedge \mathsf{is_sent}(\ell', m')$ then $m = m'$.

Then, we call $(\mathsf{MT}, \mathcal{L}, \mathcal{L}_0, \mathcal{T})$ a process template.

Asynchronous Message Passing and Counting in Presence of Byzantine Faults. In this section we introduce two ways of modeling message passing: by storing messages in sets, and by counting messages. As in [23], we do not explicitly model Byzantine processes [32], but capture their effect on the correct processes in the form of spurious messages. Although we do not discuss other kinds of faults (e.g., crashes, symmetric faults, omission faults), it is not hard to model other faults by following the modeling in [23].

We fix a set of processes Proc, which is typically defined as $\{1, \dots, n\}$ for $n \geq 1$. Further, assume that there are two disjoint sets: the set $\mathsf{Corr} \subseteq \mathsf{Proc}$ of correct processes, and $\mathsf{Byz} \subseteq \mathsf{Proc}$ of Byzantine processes (possibly empty), with $\mathsf{Byz} \cup \mathsf{Corr} = \mathsf{Proc}$. Given a process template $(\mathsf{MT}, \mathcal{L}, \mathcal{L}_0, \mathcal{T})$, we refer to $(\mathsf{MT}, \mathcal{L}, \mathcal{L}_0, \mathcal{T}, \mathsf{Corr}, \mathsf{Byz})$ as a *design*. Note that a design does not capture how processes interact with messages. To do so, in Table 1, we define message passing (MP) and message counting (MC) models as interpretations of the signature $(Msg, MsgSets, init, card, add, inTransit)$, with the following informal meaning:

- *Msg*: the set of all messages that can be exchanged by the processes,
- *MsgSets*: collections of messages,
- *init*: the empty collection of messages,
- *card*: a function that counts messages of the given type,
- *add*: a function that adds a message to a collection of messages,

– *inTransit*: a function that checks whether a message is in transit and thus can be received.

Transition Systems. Fix interpretations $(Msg_I, MsgSets_I, init_I, card_I, add_I, inTransit_I)$ for $I \in \{MP, MC\}$. Then, we define a transition system $TS^I = (S^I, S_0^I, R^I, L^I)$ of processes from Proc that communicate with respect to interpretation I. We call *message-passing system* the transition system obtained using the interpretation MP, and *message-counting system* the transition system obtained using the interpretation MC.

The set S^I contains configurations, i.e., tuples $(\mathbf{p}, \mathsf{pc}, \mathsf{rcvd}, \mathsf{sent})$ having the following properties: (a) $\mathbf{p} \in \mathbb{N}_0{}^{|\Pi|}$, (b) $\mathsf{pc} : \mathsf{Corr} \to \mathcal{L}$, (c) $\mathsf{rcvd} : \mathsf{Corr} \to MsgSets_I$, and (d) $\mathsf{sent} \in MsgSets_I$. In a configuration, for every process $p \in \mathsf{Corr}$, the values $\mathsf{pc}(p)$ and $\mathsf{rcvd}(p)$ comprise the *local view* of the process p, while the components sent and $\mathbf{p}$ comprise the *shared state* of the distributed system. A configuration $\sigma \in S^I$ belongs to the set S_0^I of initial configurations, if for each process $p \in \mathsf{Corr}$, it holds that: (a) $\sigma.\mathsf{pc}(p) \in \mathcal{L}_0$, (b) $\sigma.\mathsf{rcvd}(p) = init_I$, (c) $\sigma.\mathsf{sent} = init_I$, and (d) $\sigma.\mathbf{p} \in \mathbf{P}_{RC}$.

Definition 3.2. *The transition relation R^I contains a pair of configurations $(\sigma, \sigma') \in S^I \times S^I$, if there is a correct process $p \in \mathsf{Corr}$ that satisfies:*

1. *There exists a local transition $(\ell, \mathbf{p}, \mathbf{c}, \ell') \in \mathcal{T}$ satisfying $\sigma.\mathsf{pc}(p) = \ell$ and $\sigma'.\mathsf{pc}(p) = \ell'$ and for all m in MT, $\mathbf{c}(m) = card_I(m, \sigma'.\mathsf{rcvd}(p))$. Also, it is required that $\sigma.\mathbf{p} = \sigma'.\mathbf{p} = \mathbf{p}$.*
2. *Messages are received and sent according to the signature:*
   (a) *Process p receives no message: $\sigma'.\mathsf{rcvd}(p) = \sigma.\mathsf{rcvd}(p)$, or there is a message in transit in σ that is received in σ', i.e., there is a message $msg \in Msg_I$ satisfying:*
   $inTransit_I(msg, \sigma.\mathsf{rcvd}(p), \sigma.\mathsf{sent}) \ \wedge \ \sigma'.\mathsf{rcvd}(p) = add_I(msg, \sigma.\mathsf{rcvd}(p))$.
   (b) *The shared variable sent is changed iff process p sends a message, that is, $\sigma'.\mathsf{sent} = add_I(msg, \sigma.\mathsf{sent})$, if and only if $\neg\mathsf{is_sent}(\sigma.\mathsf{pc}(p), m)$ and $\mathsf{is_sent}(\sigma'.\mathsf{pc}(p), m)$, for every $m \in \mathsf{MT}$ and $msg \in Msg_I$ of type m.*
3. *The processes different from p do not change their local states:*
   $\sigma'.\mathsf{pc}(q) = \sigma.\mathsf{pc}(q)$ *and* $\sigma'.\mathsf{rcvd}(q) = \sigma.\mathsf{rcvd}(q)$ *for* $q \in \mathsf{Corr} \setminus \{p\}$.

The labeling function $L^I : S^I \to \mathcal{L}^{|\mathsf{Corr}|} \times \left(\mathbb{N}_0{}^{|\mathsf{MT}|}\right)^{|\mathsf{Corr}|}$ labels each configuration $\sigma \in S^I$ with the vector of control states and message counters, i.e., $L^I(\sigma) = ((\ell_1, \dots, \ell_{|\mathsf{Corr}|}), (\mathbf{c}_1, \dots, \mathbf{c}_{|\mathsf{Corr}|}))$ such that $\ell_p = \sigma.\mathsf{pc}(p)$ and $\mathbf{c}_p(m) = card_I(m, \sigma.\mathsf{rcvd}(p))$ for $p \in \mathsf{Corr}$, $m \in \mathsf{MT}$. (For simplicity we use the convention that $\mathsf{Corr} = \{1, \dots j\}$, for some $j \in \mathbb{N}$.) Note that L^I labels a configuration with the process control states and the number of messages received by each process.

The message-passing transition systems have the following features. The messages sent by *correct* processes are stored in the shared set sent. In this modeling, the messages from Byzantine processes are not stored in sent explicitly, but can be received at any step. Each correct process $p \in \mathsf{Corr}$ stores received messages

in its local set $\mathsf{rcvd}(p)$, whose elements originate from the messages stored in the set sent or from Byzantine processes.

The message-counting transition systems have the following features. Messages are not stored explicitly, but are only counted. We maintain two vectors of counters: (i) representing the number of messages that originate from correct processes (these messages have the tag C), and (ii) representing the number of messages that originate from faulty processes (these messages have the tag F). Each correct process $p \in \mathsf{Corr}$ keeps two such vectors of counters $\mathbf{c}_\mathsf{C}$ and $\mathbf{c}_\mathsf{F}$ in its local variable $\mathsf{rcvd}(p)$. In the following, we refer to $\mathbf{c}_\mathsf{C}$ and $\mathbf{c}_\mathsf{F}$ using the notation $[\mathsf{rcvd}(p)]_\mathsf{C}$ and $[\mathsf{rcvd}(p)]_\mathsf{F}$. The number of sent messages is also stored as a pair of vectors $[\mathsf{sent}]_\mathsf{C}$ and $[\mathsf{sent}]_\mathsf{F}$. By the definition of the transition relation R^{MC}, the vector $[\mathsf{sent}]_\mathsf{F}$ is always equal to the zero vector, whereas the correct process p can increment its counter $[\mathsf{rcvd}(p)]_\mathsf{F}$, if $[\mathsf{rcvd}(p)]_\mathsf{F}(m) < |\mathsf{Byz}|$, for every $m \in \mathsf{MT}$.

To prove bisimulation between a message-passing system and a message-counting system — built from the same design — we introduce the following relation on the configurations of both systems:

Definition 3.3. *Let $H^\# \subseteq S^{MP} \times S^{MC}$ such that $(\sigma, \sigma^\#) \in H^\#$ if for all processes $p \in \mathsf{Corr}$ and message types $m \in \mathsf{MT}$:*

1. $\sigma^\#.\mathsf{pc}(p) = \sigma.\mathsf{pc}(p)$
2. $\sigma^\#.[\mathsf{rcvd}(p)]_\mathsf{C}(m) = |\{q \in \mathsf{Corr}\colon \langle m, q\rangle \in \sigma.\mathsf{rcvd}(p)\}|$
3. $\sigma^\#.[\mathsf{rcvd}(p)]_\mathsf{F}(m) = |\{q \in \mathsf{Byz}\colon \langle m, q\rangle \in \sigma.\mathsf{rcvd}(p)\}|$
4. $\sigma^\#.[\mathsf{sent}]_\mathsf{C}(m) = |\{q \in \mathsf{Corr}\colon \langle m, q\rangle \in \sigma.\mathsf{sent}\}|$
5. $\sigma^\#.[\mathsf{sent}]_\mathsf{F}(m) = 0$
6. $\{q \in \mathsf{Proc}\colon \langle m, q\rangle \in \sigma.\mathsf{sent}\} \subseteq \mathsf{Corr}$
7. $\sigma.\mathsf{rcvd}(p) \subseteq \sigma.\mathsf{sent} \cup \{\langle m, q\rangle : m \in \mathsf{MT}, q \in \mathsf{Byz}\}$
8. $\mathsf{is_sent}(\sigma.\mathsf{pc}(p), m) \leftrightarrow \langle m, p\rangle \in \sigma.\mathsf{sent}$

Theorem 3.4. *For a message-passing system TS^{MP} and a message-counting system TS^{MC} defined over the same design, $H^\#$ is a bisimulation.*

The key argument to prove the Theorem 3.4 is that given a message counting state $\sigma^\#$, if a step increases a counter $\mathsf{rcvd}(p)$, in the message passing system this transition can be mirrored by receiving an arbitrary message in transit. In fact, in both systems, once a message is sent it can be received at any future step. We will see that in the timed version this argument does not work anymore, due to the restricted time interval in which a message must be received.

4 Messages with Time Constraints

We now add time constraints to both, message-passing systems and message-counting systems. Following the definitions from distributed algorithms [35,40], we assume that every message is delivered within a predefined time bound, that is, not earlier than τ^- time units and not later than τ^+ times units since the instant it was sent, with $0 \le \tau^- \le \tau^+$. We use naturals for τ^- and τ^+ for consistency with the literature on timed automata.

As can be seen from Sect. 2, to define a timed automaton, one has to provide an invariant and a switch relation. In the following, we fix the invariants and switch relations with respect to the timing constraints τ^- and τ^+ on messages. However, the specifications of distributed algorithms may refer to time, e.g., "If a correct process accepts the message *(round k)* at time t, then every correct process does so by time $t + t_{\mathrm{del}}$" [35]. Therefore, we assume that a *specification invariant* (or *user invariant*) $I_U : 2^{\mathsf{AP}} \to \Psi(U)$ and a *specification switch relation* (or *user switch relation*) $E_U : 2^{\mathsf{AP}} \times 2^{\mathsf{AP}} \to \Psi(U) \times 2^U$ are given as input. Then, we will refer to the tuple $(\mathcal{L}, \mathcal{L}_0, \mathcal{T}, \mathsf{Proc}, I_U, E_U)$ as a *timed design* and we will assume that a timed design is fixed in the following.

Using a timed design, we will use message-passing and message-counting systems to derive two timed automata. For a message of type m sent by a correct process p, the message-passing system uses a clock $c\langle m, p\rangle$ to store *the delay since the message* $\langle m, p\rangle$ *was sent.* The message-counting system stores *the delay since the* ith *message of type* m *was sent*, for all i and m. Both timed automata specify an invariant to constrain the time required to deliver a message.

Definition 4.1 (Message-passing timed automaton). *Given a message-passing system* $TS^{\mathsf{MP}} = (S^{\mathsf{MP}}, S_0^{\mathsf{MP}}, R^{\mathsf{MP}}, L^{\mathsf{MP}})$ *defined over a timed system design* $(\mathcal{L}, \mathcal{L}_0, \mathcal{T}, \mathsf{Proc}, I_U, E_U)$*, we say that a timed automaton* $TA^{\mathsf{MP}} = (S^{\mathsf{MP}}, S_0^{\mathsf{MP}}, R^{\mathsf{MP}}, L^{\mathsf{MP}}, U \cup X^{\mathsf{MP}}, I^{\mathsf{MP}}, E^{\mathsf{MP}})$ *is a* message-passing timed automaton*, if it has the following properties:*

1. *There is one clock per message that can be sent by a correct process:* $X^{\mathsf{MP}} = \{c\langle m, p\rangle : m \in \mathsf{MT}, p \in \mathsf{Corr}\}$.
2. *For each discrete transition* $(\sigma, \sigma') \in R^{\mathsf{MP}}$*, the state switch relation* $E^{\mathsf{MP}}(\sigma, \sigma')$ *ensures the specification invariant and resets the given specification clocks and the clocks corresponding to the message sent in transition* (σ, σ')*. That is, if* (φ_U, Y_U) *is the guard, and specification clocks are in* $E_U(L^{\mathsf{MP}}(\sigma), L^{\mathsf{MP}}(\sigma'))$*, then* $E^{\mathsf{MP}}(\sigma, \sigma') = (\varphi_U, Y_U \cup \{c\langle m, p\rangle : \langle m, p\rangle \in \sigma'.\mathsf{sent} \setminus \sigma.\mathsf{sent}\})$.
3. *Each state* $\sigma \in S^{\mathsf{MP}}$ *has the invariant* $I^{\mathsf{MP}}(\sigma) = I_U(L^{\mathsf{MP}}(\sigma)) \wedge \varphi^-_{\mathsf{MP}} \wedge \varphi^+_{\mathsf{MP}}$ *composed of:*
   (a) *the specification invariant* $I_U(L^{\mathsf{MP}}(\sigma))$*;*
   (b) *the lower bound on the age of received messages:* $\varphi^-_{\mathsf{MP}} = \bigwedge_{\langle m,p\rangle \in M} c\langle m, p\rangle \geq \tau^-$ *for* $M = \{\langle m, p\rangle \in \mathsf{MT} \times \mathsf{Corr} \colon \exists q \in \mathsf{Corr}.\ \langle m, p\rangle \in \sigma.\mathsf{rcvd}(q)\}$*; and*
   (c) *the upper bound on the age of messages that are in transit:* $\varphi^+_{\mathsf{MP}} = \bigwedge_{(m,p) \in M} 0 \leq c\langle m, p\rangle \leq \tau^+$ *for* $M = \{\langle m, p\rangle \in \mathsf{MT} \times \mathsf{Corr} \colon \langle m, p\rangle \in \sigma.\mathsf{sent} \setminus \bigcap_{q \in \mathsf{Corr}} \sigma.\mathsf{rcvd}(q)\}$.

Definition 4.2 (Message-counting timed automaton). *Given a message-counting system* $TS_{\mathsf{MC}} = (S^{\mathsf{MC}}, S_0^{\mathsf{MC}}, R^{\mathsf{MC}}, L^{\mathsf{MC}})$ *defined over a timed design* $(\mathcal{L}, \mathcal{L}_0, \mathcal{T}, \mathsf{Proc}, I_U, E_U)$*, we say that a timed automaton* $TA_{\mathsf{MC}} = (S^{\mathsf{MC}}, S_0^{\mathsf{MC}}, R^{\mathsf{MC}}, L^{\mathsf{MC}}, U \cup X^{\mathsf{MC}}, I^{\mathsf{MC}}, E^{\mathsf{MC}})$ *is a* message-counting timed automaton*, if it has the following properties:*

1. *There is one clock per message type and number of messages sent. That is,* $X^{\mathsf{MC}} = \{c\langle m, i\rangle : m \in \mathsf{MT}, 1 \leq i \leq |\mathsf{Corr}|\}$.

2. *For each discrete transition* $(\sigma, \sigma') \in R^{\mathsf{MC}}$*, the state switch relation* $E^{\mathsf{MC}}(\sigma, \sigma')$ *ensures the specification invariant and resets the given specification clocks and the clocks corresponding to message counters updated by* (σ, σ')*. That is, if* $(\varphi_U, Y_U) = E_U(L^{\mathsf{MC}}(\sigma), L^{\mathsf{MC}}(\sigma'))$*, then the switch relation* $E^{\mathsf{MC}}(\sigma, \sigma')$ *is* $(\varphi_U, Y_U \cup \{c\langle m, k\rangle : m \in \mathsf{MT}, k = \sigma'.\mathsf{sent}(m) = \sigma.\mathsf{sent}(m) + 1\})$*.*
3. *Each state* $\sigma \in S^{\mathsf{MC}}$ *has the invariant* $I^{\mathsf{MC}}(\sigma) = I_U(L^{\mathsf{MC}}(\sigma)) \wedge \varphi^-_{\mathsf{MC}} \wedge \varphi^+_{\mathsf{MC}}$ *composed of:*
   (a) the specification invariant $I_U(L^{\mathsf{MC}}(\sigma))$*;*
   (b) $\varphi^-_{\mathsf{MC}} = \bigwedge_{m \in \mathsf{MT}} a(m) > 0 \rightarrow c\langle m, a(m)\rangle \geq \tau^-$ *for the numbers* $a(m) = \max_{p \in \mathsf{Corr}}[\sigma.\mathsf{rcvd}(p)(m)]_{\mathsf{C}}$*. If a correct process has received* $a(m)$ *messages of type* m *from correct processes, then the* $a(m)$*-th message of type* m*, for every* $m \in \mathsf{MT}$*, was sent at least* τ^- *time units earlier.*
   (c) $\varphi^+_{\mathsf{MC}} = \bigwedge_{m \in \mathsf{MT}} \bigwedge_{b(m) < j \leq \sigma.\mathsf{sent}(m)} 0 \leq c\langle m, j\rangle \leq \tau^+$ *for the numbers* $b(m) = \min_{p \in \mathsf{Corr}}[\sigma.\mathsf{rcvd}(p)(m)]_{\mathsf{C}}$*. If there is a correct process that has received* $b(m)$ *messages of type* m *from correct processes, then for every number of messages* $j > b(m)$*, the respective clock is bounded by* τ^+*.*

While the number of employed clocks is the same, the latter model is "more abstract": by forgetting the identity of the sender, indeed, several configurations of the message-passing timed automaton can be mapped on the same configuration of the message-counting timed automaton.

5 Precision of Message Counting with Time Constraints

While Theorem 3.4 establishes a strong equivalence — that is, a bisimulation relation — between message-passing transition systems, we will show in Theorem 5.1 that message-passing timed automata and message-counting timed automata are not necessarily equivalent in the sense of timed bisimulation. Remarkably, such automata are also not necessarily equivalent in the sense of time-abstracting bisimulation. These results show an upper bound on the degree of precision achievable by model checking of timed properties of FTDAs by counting messages. Nevertheless, we show that such automata simulate each other, and thus they satisfy the same ATCTL formulas (Corollarys 5.10 and 6.2).

Theorem 5.1. *There exists a timed design whose message-passing timed automaton* TA^{MP} *and message-counting timed automaton* TA^{MC} *satisfy:*

1. *There is no initial timed bisimulation between* TA^{MP} *and* TA^{MC}*.*
2. *There is no initial time-abstracting bisimulation between* TA^{MP} *and* TA^{MC}*.*

Proof (sketch). We give an example of a timed design proving Point 2. Since timed bisimulation is a special case of time-abstracting bisimulation, this example also proves Point 1.

We use the process template shown in Fig. 4 on page 7. Formally, this template is defined as follows: there is one parameter, i.e., $\Pi = \{n\}$, one message type, i.e., $\mathsf{MT} = \{M\}$, and two control states, i.e., $\mathcal{L} = \{\ell_0, \ell_1\}$. There are two

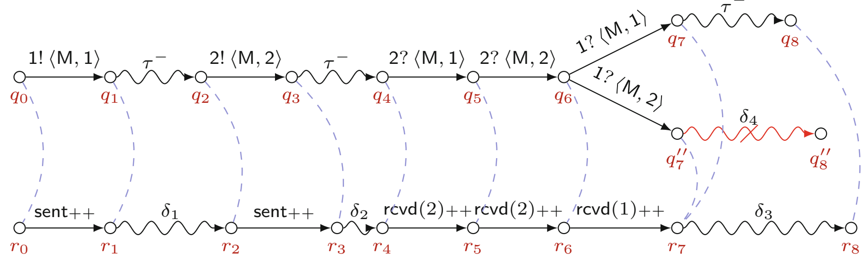


Fig. 5. Two runs of TA^{MP} (above) and one run of TA^{MC} (below) that violate time-abstracting bisimulation when $\tau^+ = 2\tau^-$. Circles and edges illustrate states and transitions. Edge labels are as follows: τ^- or δ_i designate a time step with the respective delay; i! ⟨M, j⟩ and i? ⟨M, j⟩ designate send and receive of a message ⟨M, j⟩ by process i in the message-passing system; sent++ and rcvd(i)++ designate send and receive of a message M by some process and process i respectively.

types of transitions: $t_1^{\mathbf{P}} = (\ell_0, \mathbf{p}, \mathbf{c}_4, \ell_1)$ and $t_2^{\mathbf{P}} = (\ell_1, \mathbf{p}, \mathbf{c}_5, \ell_1)$. The conditions $\mathbf{c}_4$ and $\mathbf{c}_5$ require that $\mathbf{c}_4(M) = 0$ and $\mathbf{c}_5(M) \geq 0$ respectively. Every process sends a message of type M when going from ℓ_0 to ℓ_1, i.e., $\mathsf{is_sent}(\ell, M) = \mathsf{T}$ iff $\ell = \ell_1$. Then the processes self-loop in the control state ℓ_1 (by doing so, they can receive messages from the other processes).

Consider the system of two correct processes and no Byzantine processes, that is, $\mathsf{Corr} = \{1, 2\}$ and $\mathsf{Byz} = \emptyset$. We fix the upper bound on message delays to be $\tau^+ = 2\tau^- > 0$. For the sake of this proof, we set $U = \emptyset$, and thus I_U and E_U are defined trivially. Together, these constraints define a timed design.

Figure 5 illustrates two runs of a TA^{MP} and a run of TA^{MC} that should be matched by a time-abstracting bisimulation, if one exists. We show by contradiction that no such relation exists. Note that the message $\langle M, 1\rangle$ has been received by all processes at the timed state q_7 and has not been received by the first process at the timed state q_7''. Thus the timed state q_7 admits a time step, while the timed state q_7'' does not. Indeed, on one hand, the timed automaton TA^{MP} can advance the clocks by at most $\tau^+ - \tau^- = \tau^-$ time units in q_7 before the clock attached to the message $\langle M, 2\rangle$ expires; on the other hand, in q_7'', the timed automaton TA^{MP} cannot advance the clocks before the clock attached to the message $\langle M, 1\rangle$ expires. However, both states must be time-abstract related to the state r_7 of TA^{MC}, because they both received the same number of messages of type M and thus their labels coincide, from which we derive the required contradiction. Hence, proving that there is no time-abstracting bisimulation. □

From Theorem 5.1, it follows that message counting abstraction is not precise enough to preserve an equivalence relation as strong as bisimulation. However, for abstraction-based model checking a coarser relation, namely, timed-simulation equivalence, would be sufficient. In one direction, timed-simulation is easy: a discrete configuration of a message-passing timed automaton can be mapped to the configuration of the message-counting timed automaton by just counting

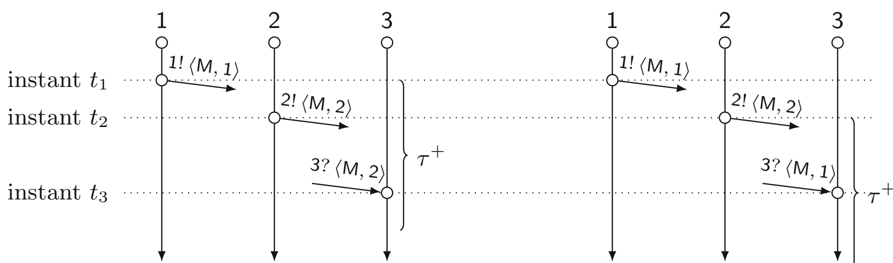


Fig. 6. Receiving messages in order relaxes constraints of delay transitions

the messages for each message type, while the clocks assignments are kept the same. The other direction is harder: A first approach would be to map a configuration of a message-counting timed automaton to all the configurations of the message-passing timed automaton, where the message counters are equal to the cardinalities of the sets of received messages. This mapping is problematic because of the interplay of message re-ordering and timing constraints:

Example 5.2 Figure 6 exemplifies a problematic behavior that originates from the interplay of message re-ordering and timing constraints on message delays. In the figure we see the space-time diagram of two timed message passing runs, where first process 1 sends $\langle M, 1\rangle$ at instant t_1, and then process 2 sends $\langle M, 2\rangle$ at a later time $t_2 > t_1$. In the run on the left, process 3 receives $\langle M, 2\rangle$ at instant t_3 and has not received $\langle M, 1\rangle$ before. In the run on the right process 3 receives $\langle M, 1\rangle$ at instant t_3. Hence, at t_3 on the left $\langle M, 1\rangle$ is in transit, while on the right $\langle M, 2\rangle$ is in transit, which has been sent after $\langle M, 1\rangle$. As indicated by the τ^+ intervals, due to the invariants from Definition 4.1[3c], the left run is more restricted: On the left within one time step the clocks can be advanced by $\tau^+ - (t_3 - t_1)$ while on the right the clocks can advance further, namely, by $\tau^+ - (t_3 - t_2) > \tau^+ - (t_3 - t_1)$. Message counting timed automata abstract away the origin of the messages, and intuitively, relate the sending of the ith message to the reception of i messages, which correspond to runs where messages are received "in order", like in the run on the right. We shall formalize this below. $\triangleleft$

In the following, we exclude from the simulation relation those states where an in-transit message has been sent before a received one, and only consider so-called well-formed states where the messages are received in the *chronological* order of the sending (according to the clocks of timed automata). Indeed, we use the fact that the timing constraints of well-formed states in the message-passing system match the timing constraints in the message-counting system.

Definition 5.3 (Well-formed state). *For a message-passing timed automaton TA^{MP} with $TS(TA^{MP}) = (Q, Q_0, \Delta, \lambda)$, a state $(s, \mu, \nu) \in Q$ is* well-formed*, if for*

each message type $m \in \mathsf{MT}$*, each process* $p \in \mathsf{Corr}$ *that has received a message* $\langle m, p' \rangle$ *has also received all messages of type* m *sent earlier than* $\langle m, p' \rangle$*:*

$$\langle m, p' \rangle \in s.\mathsf{rcvd}(p) \wedge \mu(c\,\langle m, p'' \rangle) > \mu(c\,\langle m, p' \rangle) \\ \rightarrow \langle m, p'' \rangle \in s.\mathsf{rcvd}(p) \text{ for } p', p'' \in \mathsf{Corr} \quad (1)$$

Observe that because messages can be sent at precisely the same time, there can be different well-formed states s and s' with $s.\mathsf{rcvd}(p) \neq s'.\mathsf{rcvd}(p)$. Also, considering only well-formed states does not imply that the messages are received according to the sending order in a run (which would correspond to FIFO).

We will use a mapping WF to abstract arbitrary states of any message passing timed automaton to sets of well-formed states in the same automaton.

Definition 5.4 *Given a message-passing timed automaton* TA^{MP} *with the transition system* $TS(TA^{\mathsf{MP}}) = (Q, Q_0, \Delta, \lambda)$*, we define a mapping* $\mathsf{WF} : Q \to 2^Q$ *that maps an automaton state* $(s, \mu, \nu) \in Q$ *into a set of well-formed states with each* $(s', \mu', \nu') \in \mathsf{WF}((s, \mu, \nu))$ *having the following properties:*

1. $\mu' = \mu$*,* $\nu' = \nu$*,* $s'.\mathsf{sent} = s.\mathsf{sent}$*, and* $s.\mathsf{pc}(p) = s'.\mathsf{pc}(p)$ *for* $p \in \mathsf{Corr}$*, and*
2. $|\{q \colon \langle m, q \rangle \in s'.\mathsf{rcvd}(p)\}| = |\{q \colon \langle m, q \rangle \in s.\mathsf{rcvd}(p)\}|$ *for* $m \in \mathsf{MT}, p \in \mathsf{Corr}$*.*

One can show that every timed state $q \in Q$ has at least one state in $\mathsf{WF}(q)$:

Proposition 5.5. *Let* TA^{MP} *be a message-passing timed automaton, and* $TS(TA^{\mathsf{MP}}) = (Q, Q_0, \Delta, \lambda)$*. For every state* $q \in Q$*, the set* $\mathsf{WF}(q)$ *is not empty.*

Using Proposition 5.5, one can show that the well-defined states simulate all the timed states of a message-passing timed automaton:

Theorem 5.6. *If* TA^{MP} *is a message-passing timed automaton, and if* $TS(TA^{\mathsf{MP}}) = (Q, Q_0, \Delta, \lambda)$*, then* $\{(q, r) \colon q \in Q, r \in \mathsf{WF}(q)\}$ *is an initial timed simulation.*

Theorem 5.6 suggests that timed automata restricted to well-formed states might help us in avoiding the negative result of Theorem 5.1. To this end, we introduce a *well-formed message-passing timed automaton*. Before that, we note that Eq. (1) of Definition 5.3 can be transformed to a state invariant. We denote such a state invariant as I^{WF}.

Definition 5.7 (Well-formed MPTA). *Given a message-passing timed automaton* $TA^{\mathsf{MP}} = (S, S_0, R, L, U \cup X, I, E)$*, its well-formed restriction* $TA^{\mathsf{MP}}_{\mathsf{WF}}$ *is the timed automaton* $(S, S_0, R, L, U \cup X, I \wedge I^{\mathsf{WF}}, E)$*.*

Since the well-formed states are included in the set of timed states, and the well-formed states simulate timed states (Theorem 5.6), we obtain the following:

Corollary 5.8. *Let* TA^{MP} *be a message-passing timed automaton and* $TA^{\mathsf{MP}}_{\mathsf{WF}}$ *be its well-formed restriction. These timed automata are timed-simulation equivalent:* $TA^{\mathsf{MP}} \simeq^t TA^{\mathsf{MP}}_{\mathsf{WF}}$*.*

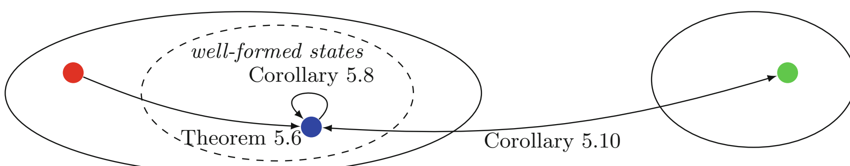


Fig. 7. Simulations constructed in Theorems 5.6–5.10. Small circles depict states of the transition systems. An arrow from a state s to a state t illustrates that the pair (s, t) belongs to a timed simulation

As a consequence of Theorems 3.4, 5.6, and Corollary 5.8, one obtains that there is a timed bisimulation equivalence between a well-formed message-passing timed automaton and the corresponding message-counting timed automaton, which is obtained by forgetting the sender of the messages and just counting the sent and delivered messages.

Theorem 5.9. *Let TA^{MP} be a message-passing timed automaton and TA^{MC} be a message-counting timed automaton defined over the same timed system design. Further, let $TA^{\mathsf{MP}}_{\mathsf{WF}}$ be the well-formed restriction of TA^{MP}. There exists an initial timed bisimulation: $TA^{\mathsf{MP}}_{\mathsf{WF}} \approx^t TA^{\mathsf{MC}}$.*

By collecting Theorem 5.9 and Corollary 5.8 we conclude that there is a timed simulation equivalence between MPTA and MCTA:

Corollary 5.10. *Let TA^{MP} be a message-passing timed automaton and TA^{MC} be a message-counting timed automaton defined over the same timed system design. TA^{MP} and TA^{MC} are timed-simulation equivalent: $TA^{\mathsf{MP}} \simeq^t TA^{\mathsf{MC}}$.*

Figure 7 uses arrows to depict the timed simulations presented in this work.

6 Conclusions

Asynchronous Systems. For systems considered in Sect. 3, we conclude from Theorem 3.4 that message-counting systems are detailed enough for model checking of properties written in CTL^*:

Corollary 6.1. *For a $\mathsf{CTL}^\star$ formula φ, a message-passing system TS^{MP} and a message-counting system TS^{MC} defined over the same design, $TS^{\mathsf{MP}} \models \varphi$ if and only if $TS^{\mathsf{MC}} \models \varphi$.*

The corollary implies that the message counting abstraction does not introduce spurious behavior. In contrast, data and counter abstractions introduced in [22] may lead to spurious behavior as only simulation relations have been shown for these abstractions.

Timed Systems. For systems considered in Sect. 4, we consider specifications in the temporal logic ATCTL [14], which restricts TCTL [6] as follows: first, negations only appear next to propositions $p \in \mathsf{AP} \cup \Psi(U)$, and second, the temporal operators are restricted to $\mathsf{A\,F}_{\sim c}$, $\mathsf{A\,G}_{\sim c}$, and $\mathsf{A\,U}_{\sim c}$.

To derive that message-counting timed automata are sufficiently precise for model checking of ATCTL formulas (in the following corollary), we combine the following results: (i) Simulation-equivalent systems satisfy the same formulas of ACTL, e.g. see [11, Theorem 7.76]; (ii) Reduction of TCTL model checking to CTL model checking by clock embedding [11, p. 706]; (iii) Corollary 5.10.

Corollary 6.2. *For a message-passing timed automaton TA^{MP} and a message-counting timed automaton TA^{MC} defined over the same timed design and an* ATCTL*-formula φ, the following holds: $TA^{\mathsf{MP}} \models \varphi$ if and only if $TA^{\mathsf{MC}} \models \varphi$.*

Future Work. Most of the timed specifications of interest for FTDAs (e.g., fault-tolerant clock synchronization algorithms [35,39,40]) are examples of time-bounded specifications, thus belonging to the class of timed safety specifications. These algorithms can be encoded as message-passing timed automata (Definition 4.1). In this paper, we have shown that model checking of these algorithms can also be done at the level of message-counting timed automata (Definition 4.2). Based on this it appears natural to apply the abstraction-based parameterized model checking technique from [22]. However, we are still facing the challenge of having a parameterized number of clocks in Definition 4.2. We are currently working on another abstraction that addresses this issue. This will eventually allow us to do parameterized model checking of timed fault-tolerant distributed algorithms using UPPAAL [12] as back-end model checker.

Related Work. As discussed in [23], while modeling message passing is natural for fault-tolerant distributed algorithms (FTDAs), message counting scales better for asynchronous systems, and also builds a basis for efficient parameterized model checking techniques [22,28]. We are interested in corresponding results for timed systems, that is, our long-term research goal is to build a framework for the automatic verification of timed properties of FTDAs. Such kind of properties are particularly relevant for the analysis of distributed clock synchronization protocols [35,39,40]. This investigation combines two research areas: (i) verification of FTDAs and (ii) parameterized model checking (PMC) of timed systems.

To the best of our knowledge, most of the existing literature on (i) can model only the discrete behaviors of the algorithms themselves [4,5,18,20,22,28,38]. Consequently they can neither reason about nor verify their timed properties. This motivated us to extend existing techniques for modeling and abstracting FTDAs, such as message passing and message counting systems together with the message counting abstraction, to timed systems.

Most of the results about PMC of timed systems [1–3,8–10,31,34] are restricted to systems whose interprocess communication primitives have other systems in mind than FTDAs. For instance, the local state space is fixed and finite and independent of the parameters, while message counting in FTDAs

requires that the local state space depends on the parameters. This motivated us to introduce the notions of *message passing timed automata* and *message counting timed automata*. Besides, the literature typically focuses on decidability, e.g., [1,3,9,34] analyze decidability for different variants of the parameterized model checking problem (e.g., integer vs. continuous time, safety vs. liveness, presence vs. absence of controller). Our work focuses on establishing relations between different timed models, with the goal of using these relations for abstraction-based model checking.

References

1. Abdulla, P.A., Deneux, J., Mahata, P.: Multi-clock timed networks. In: LICS, pp. 345–354 (2004)
2. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35873-9_28
3. Abdulla, P.A., Jonsson, B.: Model checking of systems with many identical timed processes. Theor. Comput. Sci. **290**(1), 241–264 (2003)
4. Alberti, F., Ghilardi, S., Orsini, A., Pagani, E.: Counter abstractions in model checking of distributed broadcast algorithms: some case studies. In: CILC, pp. 102–117 (2016)
5. Alberti, F., Ghilardi, S., Pagani, E.: Counting constraints in flat array fragments. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 65–81. Springer, Heidelberg (2016). doi:10.1007/978-3-319-40229-1_6
6. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: LICS, pp. 414–425 (1990)
7. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
8. Aminof, B., Kotek, T., Rubin, S., Spegni, F., Veith, H.: Parameterized model checking of rendezvous systems. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 109–124. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44584-6_9
9. Aminof, B., Rubin, S., Zuleger, F., Spegni, F.: Liveness of parameterized timed networks. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 375–387. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47666-6_30
10. Außerlechner, S., Jacobs, S., Khalimov, A.: Tight cutoffs for guarded protocols with fairness. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 476–494. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49122-5_23
11. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Massachusetts (2008)
12. Behrmann, G., David, A., Larsen, K.G., Hakansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST, pp. 125–126 (2006)
13. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM **32**(4), 824–840 (1985)
14. Bulychev, P., Chatain, T., David, A., Larsen, K.G.: Efficient on-the-fly algorithm for checking alternating timed simulation. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 73–87. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04368-0_8

15. Čerāns, K.: Decidability of bisimulation equivalences for parallel timer processes. In: Bochmann, G., Probst, D.K. (eds.) CAV 1992. LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993). doi:10.1007/3-540-56496-9_24
16. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Massachusetts (1999)
17. 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)
18. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 161–181. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54013-4_10
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
20. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 315–331. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_22
21. Függer, M., Schmid, U.: Reconciling fault-tolerant distributed computing and systems-on-chip. Distrib. Comput. **24**(6), 323–355 (2012)
22. 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)
23. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards modeling and model checking fault-tolerant distributed algorithms. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 209–226. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39176-7_14
24. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: The Theory of Timed I/O Automata. Morgan & Claypool Publishers, San Rafael (2006)
25. Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL 2017. (to appear, preliminary version at arXiv:1608.05327)
26. Konnov, I., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 125–140. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44584-6_10
27. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 85–102. Springer,
Heidelberg (2015). doi:10.1007/978-3-319-21690-4_6
28. Konnov, I., Veith, H., Widder, J.: What you always wanted to know about model checking of fault-tolerant distributed algorithms. In: Mazzara, M., Voronkov, A. (eds.) PSI 2015. LNCS, vol. 9609, pp. 6–21. Springer, Heidelberg (2016). doi:10.1007/978-3-319-41579-6_2
29. Lynch, N., Vaandrager, F.: Forward and backward simulations for timing-based systems. In: Bakker, J.W., Huizing, C., Roever, W.P., Rozenberg, G. (eds.) REX 1991. LNCS, vol. 600, pp. 397–446. Springer, Heidelberg (1992). doi:10.1007/BFb0032002
30. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN, pp. 541–550 (2003)

31. Namjoshi, K.S., Trefler, R.J.: Uncovering symmetries in irregular process networks. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 496–514. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35873-9_29
32. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
33. Song, Y.J., Renesse, R.: Bosco: one-step byzantine asynchronous consensus. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 438–450. Springer, Heidelberg (2008). doi:10.1007/978-3-540-87779-0_30
34. Spalazzi, L., Spegni, F.: Parameterized model-checking of timed systems with conjunctive guards. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 235–251. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12154-3_15
35. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. J. ACM **34**(3), 626–645 (1987)
36. Srikanth, T.K., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distrib. Comput. **2**, 80–94 (1987)
37. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. FMSD **18**, 25–68 (2001)
38. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distrib. Comput. **23**(5–6), 341–358 (2011)
39. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distrib. Comput. **20**(2), 115–140 (2007)
40. Widder, J., Schmid, U.: The theta-model: achieving synchrony without clocks. Distrib. Comput. **22**(1), 29–47 (2009)

Chapter 3

Parameterized model checking of fault-tolerant distributed algorithms by abstraction

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. FMCAD, pp. 201–209, 2013.

DOI: http://dx.doi.org/10.1109/FMCAD.2013.6679411

Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, Josef Widder
Vienna University of Technology (TU Wien)

***Abstract*—We introduce an automated parameterized verification method for fault-tolerant distributed algorithms (FTDA). FTDAs are parameterized by both the number of processes and the assumed maximum number of faults. At the center of our technique is a parametric interval abstraction (PIA) where the interval boundaries are arithmetic expressions over parameters. Using PIA for both data abstraction and a new form of counter abstraction, we reduce the parameterized problem to finite-state model checking. We demonstrate the practical feasibility of our method by verifying safety and liveness of several fault-tolerant broadcasting algorithms, and finding counter examples in the case where there are more faults than the FTDA was designed for.**

I. INTRODUCTION

Fault-tolerant distributed algorithms (FTDA) constitute a core topic of distributed algorithm theory, with a rich body of results [27], [2]. Yet, they have not been systematically studied from a model checking point of view. For FTDAs one typically considers systems of n processes out of which at most t may be faulty. In this paper we consider various faults such as crash faults, omissions, and Byzantine faults. As FTDAs are parameterized in n and t, we require parameterized verification to establish the correctness of an FTDA. The pragmatic approach to verify a system of *fixed* size is not practical, as only very small instances can be verified due to state space explosion [24], [36], [34]. While in classic parameterized model checking the number of processes n is the sole parameter, for FTDAs, t is also a parameter, and is essentially a fraction of n, expressed by a *resilience condition*, e.g., $n > 3t$. Thus, one has to reason about all runs with $n-f$ non-faulty and f faulty processes, where $f \leq t$ and $n > 3t$.

From an operational viewpoint, FTDAs typically consist of multiple processes that communicate by passing messages. As senders can be faulty, a receiver cannot wait for a message from a specific sender process. Thus, most FTDAs use counters to reason about the environment; e.g., if a process receives a certain message from more than t distinct senders, then one of the senders must be non-faulty. A large class of FTDAs expresses these counting arguments using *threshold guards:*

```
if received <m> from t+1 distinct processes
then action(m);
```

Threshold guards generalize existential and universal guards [16], i.e., rules that wait for messages from at least one or all processes, respectively. As can be seen from the above example, and as discussed in [24], existential and universal guards are not sufficient to capture advanced FTDAs: Threshold guards are a basic building block that has been used in various environments (various degrees of synchrony, fault assumptions, etc.) and FTDAs, such as consensus [15], software and hardware clock synchronization [32], [19], approximate agreement [14], and k-set agreement [13]. The ability to efficiently reason about these guards is thus a keystone for automated parameterized verification of such algorithms.

This paper considers parameterized verification of FTDAs with threshold guards and resilience conditions. We introduce a framework based on a new form of control flow automata that captures the semantics of threshold-guarded FTDAs, and propose a novel two-step abstraction technique. It is based on *parametric interval abstraction* (PIA), a generalization of interval abstraction where the interval borders are expressions over *parameters* rather than constants. Using the PIA domain, we obtain a finite-state model checking problem in two steps:

Step 1: PIA data abstraction. We evaluate the threshold guards over the parametric intervals. Thus, we abstract away unbounded variables and parameters from the process code. We obtain a parameterized system where the replicated processes are finite-state and independent of the parameters.

Step 2: PIA counter abstraction. We use a new form of counter abstraction where the process counters are abstracted to PIA. As Step 1 guarantees that we need only finitely many counters, PIA counter abstraction yields a finite-state system.

To evaluate the precision of our abstractions, we implemented our abstraction technique in a tool chain, and conducted experiments on several FTDAs. Our experiments showed the need for abstraction refinement to deal with spurious counterexamples [7] that are due to parameterized abstraction and fairness. This required novel refinement techniques, which we also discuss in this paper. In addition to refinement of PIA counter abstraction, which is automated in a loop using a model checker and an SMT solver, we are also exploiting simple user-provided invariant candidates (as in [28], [35]) to refine the abstraction.

We verify several FTDAs that have been derived from the well-known distributed broadcast algorithm by Srikanth and Toueg [32], [33], and a folklore reliable broadcasting algorithm [2, Sect. 8.2.5.1]. Each of these FTDAs tolerates different faults (e.g., crash, omission, Byzantine), and uses different threshold guards. To the best of our knowledge, we are the first to achieve parameterized automated verification of Byzantine FTDAs.

Supported by the Austrian National Research Network S11403 and S11405 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) grants PROSEED, ICT12-059, and VRG11-005. Details that had to be omitted from this paper can be found in [23].

ISBN 978-0-9835678-3-7/13. **Copyright owned jointly by the authors and FMCAD Inc.**

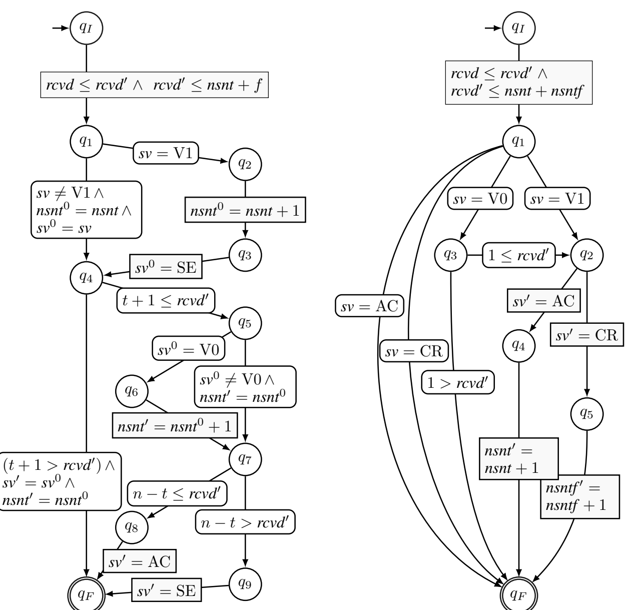


Fig. 1. CFA of our case study for Byzantine faults.

Fig. 2. CFA of FTDA from [18] (if x' is not assigned, then $x' = x$).

II. Our Approach at a Glance

To give an intuition of our method, we start with the control flow automaton (CFA) given in Figure 1 that formalizes our case study FTDA. The CFA uses the shared integer variable *nsnt* (capturing the number of messages sent by non-faulty processes), the local integer variable *rcvd* (storing the number of messages received by the process so far), and the local status variable *sv*, which ranges over a finite domain (capturing the local progress w.r.t. the FTDA). In [24] we show that this formalization captures the logic of our case study FTDA.

We use the CFA to represent one atomic *step* of the FTDA: Each edge is labeled with a guard. A path from q_I to q_F induces a conjunction of all the guards along it, and imposes constraints on the variables before the step (e.g., *sv*), after the step (sv'), and temporary variables (sv^0). If one fixes the variables before the step, different valuations (of the primed variables) that satisfy the constraints capture non-determinism.

A system consists of $n - f$ processes that concurrently execute the code corresponding to the CFA, and communicate via *nsnt*. Thus, there are two sources of unboundedness: first, the integer variables, and second, the parametric number of processes. We deal with these two issues in two steps.

Step 1: PIA data abstraction. We observe that the CFA contains several transitions which are labeled with *threshold guards* that refer to (unbounded) variables and parameters. For instance, the CFA in Figure 1 contains the following transition, which is labeled with a threshold guard:

$$q_4 \xrightarrow{\;t + 1 \leq rcvd'\;} q_5$$

The CFA also contains a guard $n - t \leq rcvd'$. Actually, the correctness of the FTDA is based on the fact that the values of the thresholds, e.g., $t+1$ and $n-t$, are sufficiently far apart from each other under the resilience condition $n > 3t \wedge f \leq t$; in particular, $(n - t) - f \geq t + 1$. These properties are also used in the manual proofs [33]. We observe that such FTDAs are designed by carefully choosing the thresholds and the resilience condition. Consequently, our abstraction must be sufficiently precise to preserve the relationship between thresholds and the resilience condition.

The second important observation is that it is not necessary to keep track of the precise value of variables that are compared against thresholds, e.g., $rcvd'$. Rather, in our case study, it is sufficient to know whether $rcvd'$ lies in the interval $[0, t+1[$, or $[t+1, n-t[$, or $[n-t, \infty[$, in order to determine which of the threshold guards of the CFA are satisfied. Our *parametric interval abstraction* PIA exploits this idea. In addition, in Step 2 we will see that we also have to distinguish 0 from other values. Thus, PIA consists of mapping integers to a finite domain of four intervals $I_0 = [0, 1[$ and $I_1 = [1, t+1[$ and $I_2 = [t+1, n-t[$ and $I_3 = [n-t; \infty[$.

Then, we replace the guards that refer to unbounded variables and parameters by their existential abstraction. For instance, the above transition with the guard "$t + 1 \leq rcvd'$" means that $rcvd'$ lies in the intervals $[t+1, n-t[$ or $[n-t, \infty[$. As these correspond to the abstract intervals I_2 and I_3, respectively, we can replace the guard by:

$$q_4 \xrightarrow{\;rcvd' = I_2 \vee rcvd' = I_3\;} q_5$$

The abstraction of the guard "$nsnt^0 = nsnt + 1$" can be expressed similarly, as later discussed in Figure 4. The expression "$rcvd' \leq nsnt + f$", which is also used in a guard, is more complicated as it involves two variables and a parameter. Still, the basic abstraction idea is the same. The corresponding abstract expression has the form $(rcvd' = I_0 \wedge nsnt = I_0) \vee (rcvd' = I_0 \wedge nsnt = I_1) \vee \cdots \vee (rcvd' = I_3 \wedge nsnt = I_3)$.

These abstract guards are Boolean expressions over equalities between variables and abstract values. Therefore, it is sufficient to interpret the variables *nsnt* and *rcvd* over the finite domain. Hence, all variables range over finite domains, and we arrive at finite state processes in this way. Our system, however, is still parameterized, namely, in the number of processes.

Step 2: PIA counter abstraction. We reduce this system to a finite state system using the following two ideas. First, we change to a counter-based representation, i.e., the global state is represented by the (abstract) shared variable *nsnt*, and by one counter for each of the local states. A counter stores how many processes are in the corresponding local state. Second, as processes interact only via the *nsnt* variable, precisely counting processes in certain states may not be necessary; as *nsnt* already ranges over the abstract domain, it is natural to count processes in terms of the same abstract domain.

The local state of a process is determined by the values of *sv* and *rcvd*. Thus, we denote by $\kappa[x, y] = I$ that the number of processes with $sv = x$ and $rcvd = y$ lies in the abstract interval I. Then, in Figure 3, the state s_0 represents the initial states with $t + 1$ to $n - t - 1$ processes having $sv = \texttt{V0}$ and 1 to t processes having $sv = \texttt{V1}$. (We omit local states that have the counter value I_0 to facilitate reading.)

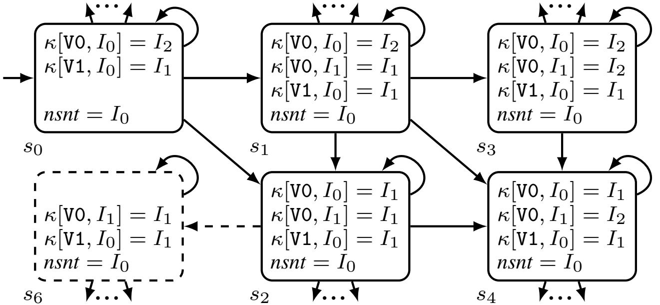


Fig. 3. A small part of the transition system obtained by counter abstraction. As shown by our experimental data in Table I of Section VII, the reachable state space is substantially larger.

Figure 3 gives a small part of the transition system obtained from the counter abstraction starting from initial state s_0. Each transition corresponds to one process taking a step in the concrete system. For instance, in the transition (s_0, s_2) a process with local state $[\mathtt{V0}, I_0]$ changes its state to $[\mathtt{V0}, I_1]$. Therefore, the counter $\kappa[\mathtt{V0}, I_0]$ is decremented and the counter $\kappa[\mathtt{V0}, I_1]$ is incremented. However, as we interpret counters over the abstract domain, the operations of incrementing and decrementing a counter are actually non-deterministic. Consequently, the transition (s_0, s_1) captures the same concrete local step as (s_0, s_2). In (s_0, s_1), the non-deterministic decrement of the abstract counter $\kappa[\mathtt{V0}, I_0]$ did not change its value.

Typically, the specifications of FTDAs refer to global states where "there is a process in a given local state" or "all processes are in a given local state." To express this via counters, we have to check whether counter values are I_0.

Abstraction refinement. Our abstraction steps result in a system which is an over-approximation of all systems with fixed parameters. For instance, the non-determinism in the counters may "increase" or "decrease" the number of processes in a system, although in all concrete system the number of processes is constant: Consider the transition (s_2, s_6) in Figure 3, and let x, y, z be the non-negative integers that are in s_2 abstracted to $\kappa[\mathtt{V0}, I_0]$, $\kappa[\mathtt{V0}, I_1]$, and $\kappa[\mathtt{V1}, I_0]$, respectively. Similarly y' and z' are abstracted to $\kappa[\mathtt{V0}, I_1]$ and $\kappa[\mathtt{V1}, I_0]$ in s_6. If the following inequalities do not have a solution under the resilience condition ($n > 3t, t \geq f$), then there is no concrete system with a transition between two states that are abstracted to s_2 and s_6, respectively.

$$1 \leq x < t+1,\ 1 \leq y < t+1,\ 1 \leq z < t+1,$$
$$1 \leq y' < t+1,\ 1 \leq z' < t+1,$$
$$x + y + z = y' + z' = n - f.$$

We use an SMT solver for this, and examine each transition of a counterexample returned by a model checker. If a transition is spurious, then we remove it from the abstract system.

Related abstractions. Interval abstraction [10] is a natural solution to the problem of unboundedness of local variables. However, if we fixed the interval bounds to numeric values, then they would not be aligned to the thresholds, and the abstraction would not be sufficiently precise to do parametric verification. At the same time, we do not have to deal with symbolic ranges over *variables* in the sense of [30], because for FTDAs the interval bounds are *constant* in each run.

Further, we want to produce a single process skeleton that is independent of parameters and captures the behavior of *all* process instances. This can be done by using ideas from existential abstraction [9], [12], [25] and sound abstraction of fairness constraints [25]. We combine these two ideas to arrive at PIA data abstraction.

The PIA counter abstraction is similar to [29], in that counters range over an abstract domain, and increment and decrement is done using existential abstraction. The domain in [29] consists of three values representing 0, 1, or *more*. This domain is sufficient for mutual-exclusion-like problems: It allows to distinguish good from bad states, while it is not possible (and also not necessary) to distinguish two bad states: A bad state is one where at least two processes are in the critical section, which is precisely abstracted in the three-valued domain. However, two bad states where, e.g., 2 and 3 processes are in the critical section, respectively, cannot be distinguished. Verification of threshold-based FTDAs requires more involved counting; e.g., we have to capture whether at least $n - t$ processes or at most t processes incremented *nsnt*. Therefore, we use counters from the PIA domain.

III. System Model with Multiple Parameters

In this section we develop all notions that are required to precisely state the parameterized model checking problem for multiple parameters. As running example, we use the parameters mentioned above, namely, the number of processes n, the upper bound on the number of faults t, and the actual number of faults f. We start to define parameterized processes (that access shared variables) in a way that allows us to modularly compose them into a parameterized system instance.

We apply this modeling to verify FTDAs as follows: as input we take a process description that uses the parameters n and t in the code. From this we construct a system instance parameterized with n, t, and f, which then describes all runs of an algorithm in which exactly f faults occur. The verification problem for a distributed algorithm in the *concrete case* with fixed n and t is the composition of model checking problems that differ in the actual value of $f \leq t$. This modeling also allows us to set $f = t + 1$, which models runs in which more faults occur than expected, and search for counterexamples. For the *parameterized case*, we introduce a resilience condition on these parameters, and require to verify the algorithm for all values of parameters that satisfy the resilience condition.

We define the parameters, local variables of the processes, and shared variables referring to a single *domain* D that is totally ordered and has the operations of addition and subtraction. In this paper we assume that D is the set of nonnegative integers $\mathbb{N}_0$.

We start with some notation. Let Y be a finite set of variables ranging over D. We denote by $D^{|Y|}$, the set of all $|Y|$-tuples of variable values. Given $\mathbf{s} \in D^{|Y|}$, we use the expression $\mathbf{s}.y$, to refer to the value of a variable $y \in Y$ in

vector $\mathbf{s}$. For two vectors $\mathbf{s}$ and $\mathbf{s}'$, by $\mathbf{s} =_X \mathbf{s}'$ we denote the fact that for all $x \in X$, $\mathbf{s}.x = \mathbf{s}'.x$ holds.

Process. The set of variables V is $\{sv\} \cup \Lambda \cup \Gamma \cup \Pi$: The variable *sv* is the *status variable* that ranges over a finite set *SV* of *status values*. The finite set Λ contains variables that range over the domain D. The variable *sv* and the variables from Λ are *local variables*. The finite set Γ contains the *shared variables* that range over D. The finite set Π is a set of *parameter variables* that range over D, and the *resilience condition RC* is a predicate over $D^{|\Pi|}$. In our example, $\Pi = \{n, t, f\}$, and the resilience condition $RC(n, t, f)$ is $n > 3t \wedge f \leq t \wedge t > 0$. Then, we denote the set of *admissible parameters* by $\mathbf{P}_{RC} = \{\mathbf{p} \in D^{|\Pi|} \mid RC(\mathbf{p})\}$.

A process operates on states from the set $S = SV \times D^{|\Lambda|} \times D^{|\Gamma|} \times D^{|\Pi|}$. Each process starts its computation in an initial state from a set $S^0 \subseteq S$. A relation $R \subseteq S \times S$ defines *transitions* from one state to another, with the restriction that the values of parameters remain unchanged, i.e., for all $(\mathbf{s}, \mathbf{t}) \in R$, $\mathbf{s} =_\Pi \mathbf{t}$. Then, a *parameterized process skeleton* is a tuple $\mathsf{Sk} = (S, S^0, R)$.

We get a process instance by fixing the parameter values $\mathbf{p} \in D^{|\Pi|}$: one can restrict the set of process states to $S|_\mathbf{p} = \{\mathbf{s} \in S \mid \mathbf{s} =_\Pi \mathbf{p}\}$ as well as the set of transitions to $R|_\mathbf{p} = R \cap (S|_\mathbf{p} \times S|_\mathbf{p})$. Then, a *process instance* is a process skeleton $\mathsf{Sk}|_\mathbf{p} = (S|_\mathbf{p}, S^0|_\mathbf{p}, R|_\mathbf{p})$ where $\mathbf{p}$ is constant.

System Instance. For fixed admissible parameters $\mathbf{p}$, a distributed system is modeled as an asynchronous parallel composition of identical processes $\mathsf{Sk}|_\mathbf{p}$. The number of processes depends on the parameters. To formalize this, we define the size of a system (the number of processes) using a function $N \colon \mathbf{P}_{RC} \to \mathbb{N}_0$, for instance, when modeling only correct processes explicitly, we use $n - f$ for $N(n, t, f)$.

Given $\mathbf{p} \in \mathbf{P}_{RC}$, and a process skeleton $\mathsf{Sk} = (S, S^0, R)$, a system instance is defined as an asynchronous parallel composition of $N(\mathbf{p})$ process instances, indexed by $i \in \{1, \ldots, N(\mathbf{p})\}$, with standard interleaving semantics. Let AP be a set of atomic propositions. A *system instance* $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a Kripke structure $(S_I, S_I^0, R_I, \mathrm{AP}, \lambda_I)$ where:

- $S_I = \{(\sigma[1], \ldots, \sigma[N(\mathbf{p})]) \in (S|_\mathbf{p})^{N(\mathbf{p})} \mid \forall i, j \in \{1, \ldots, N(\mathbf{p})\}, \sigma[i] =_{\Gamma \cup \Pi} \sigma[j]\}$ is the set of *(global) states*. Informally, a global state σ is a Cartesian product of the state $\sigma[i]$ of each process i, with identical values of parameters and shared variables at each process.
- $S_I^0 = (S^0)^{N(\mathbf{p})} \cap S_I$ is the set of *initial (global) states*, where $(S^0)^{N(\mathbf{p})}$ is the Cartesian product of initial states of individual processes.
- A transition (σ, σ') from a global state $\sigma \in S_I$ to a global state $\sigma' \in S_I$ belongs to R_I iff there is an index i, $1 \leq i \leq N(\mathbf{p})$, such that:
  - (MOVE) The i-th process *moves*: $(\sigma[i], \sigma'[i]) \in R|_\mathbf{p}$.
  - (FRAME) The values of the local variables of the other processes are preserved: for every process index $j \neq i$, $1 \leq j \leq N(\mathbf{p})$, it holds that $\sigma[j] =_{\{sv\} \cup \Lambda} \sigma'[j]$.
- $\lambda_I : S_I \to 2^{\mathrm{AP}}$ is a state labeling function.

Remark 1: The set of global states S_I and the transition relation R_I are preserved under every transposition $i \leftrightarrow j$ of process indices i and j in $\{1, \ldots, N(\mathbf{p})\}$. That is, every system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *fully symmetric* by construction.

Atomic Propositions. We define the set of atomic propositions AP to be the disjoint union of AP_{SV} and AP_D: The set AP_{SV} contains propositions that capture comparison against a given status value $Z \in SV$, i.e., $[\forall i.\ sv_i = Z]$ and $[\exists i.\ sv_i = Z]$. Further, the set of atomic propositions AP_D captures comparison of variables x, y, and a linear combination c of parameters from Π; AP_D consists of propositions of the form $[\exists i.\ x_i + c < y_i]$ and $[\forall i.\ x_i + c \geq y_i]$.

The labeling function λ_I of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ maps a state σ to expressions p from AP as follows (the existential case is defined accordingly using disjunctions):

$$[\forall i.\ sv_i = Z] \in \lambda_I(\sigma) \text{ iff } \bigwedge_{i=1}^{N(\mathbf{p})} (\sigma[i].sv = Z)$$

$$[\forall i.\ x_i + c \geq y_i] \in \lambda_I(\sigma) \text{ iff } \bigwedge_{i=1}^{N(\mathbf{p})} (\sigma[i].x + c(\mathbf{p}) \geq \sigma[i].y)$$

Temporal Logic. We specify properties using temporal logic $\mathsf{LTL}_{\text{-}\mathsf{X}}$ over AP_{SV}. We use the standard definitions of paths and $\mathsf{LTL}_{\text{-}\mathsf{X}}$ semantics [6]. A formula of $\mathsf{LTL}_{\text{-}\mathsf{X}}$ is defined inductively as: (i) a literal p or $\neg p$, where $p \in \mathrm{AP}_{SV}$, or (ii) $\mathbf{F}\,\varphi$, $\mathbf{G}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi \vee \psi$, and $\varphi \wedge \psi$, where φ and ψ are $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formulas.

Fairness. We are interested in verifying safety and liveness properties. The latter can be usually proven only in the presence of fairness constraints. As in [25], [29], we consider verification of safety and liveness in systems with *justice* fairness constraints. We define fair paths of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ using a set of justice constraints $J \subseteq \mathrm{AP}_D$. A path π of a system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *J-fair* iff for every $p \in J$ there are infinitely many states σ in π with $p \in \lambda_I(\sigma)$. By $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$ we denote that the formula φ holds on all J-fair paths of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$.

Definition 2: Given a system description containing

- a domain D,
- a parameterized process skeleton $\mathsf{Sk} = (S, S_0, R)$,
- a resilience condition *RC* (generating a set of admissible parameters $\mathbf{P}_{RC}$),
- a system size function N,
- justice requirements J,

and an $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formula φ, the *parameterized model checking problem* (PMCP) is to verify $\forall \mathbf{p} \in \mathbf{P}_{RC}.\ \mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$.

IV. Threshold-guarded FTDAs

In [24], we formalized threshold-guarded FTDAs in Promela. In order to introduce our abstraction technique, we propose a language-independent approach that focuses on the control flow and is based on control flow automata (CFA) [21].

A *guarded control flow automaton* (CFA) is an edge-labeled directed acyclic graph $\mathcal{A} = (Q, q_I, q_F, E)$ with a finite set Q of nodes called locations, an initial location $q_I \in Q$, and a final location $q_F \in Q$. A path from q_I to q_F is used to describe *one step* of a distributed algorithm. The edges have the form

$E \subseteq Q \times \texttt{guard} \times Q$, where guard is defined as an expression of one of the following forms where $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, and $\Pi = \{p_1, \ldots, p_{|\Pi|}\}$:

- if $Z \in SV$, then $sv = Z$ and $sv \neq Z$ are *status guards*;
- if x is a variable in D and $\lhd \in \{\leq, >\}$, then
$$a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i \ \lhd \ x$$
is a *threshold guard*;
- if $y, z_1, \ldots, z_k$ are variables in D for $k \geq 1$, and $\lhd \in \{=, \neq, <, \leq, >, \geq\}$, and $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, then
$$y \ \lhd \ z_1 + \cdots + z_k + \big(a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i\big)$$
is a *comparison guard*;
- a conjunction $g_1 \wedge g_2$ of guards g_1 and g_2 is a guard.

Status guards are used to capture the basic control flow. Threshold guards capture the core primitive of the FTDAs we consider. Finally, comparison guards are used to model send and receive operations. Figure 1 shows an example CFA with $\Gamma = \{nsnt\}$, $\Lambda = \{rcvd\}$, and $\Pi = \{n, t, f\}$.

Obtaining a Skeleton from a CFA. One step of a process skeleton is defined by a path from q_I to q_F in a CFA. Given *SV*, Λ, Γ, Π, *RC*, and a CFA $\mathcal{A}$, we define the process skeleton $\mathsf{Sk}(\mathcal{A}) = (S, S^0, R)$ induced by $\mathcal{A}$ as follows: The set of variables used by the CFA is $W \supseteq \Pi \cup \Lambda \cup \Gamma \cup \{sv\} \cup \{x' \mid x \in \Lambda \cup \Gamma \cup \{sv\}\}$, which may contain also temporary variables. A variable x corresponds to the value before a step, x' to the value after the step, and $x^0, x^1, \ldots$ to intermediate values. A path p from q_I to q_F induces a conjuction of all the guards along it. We call a mapping v from W to the values from the respective domains a *valuation*. We write $v \models p$ to denote that the valuation v satisfies the guards of the path p. We define the mapping between a CFA $\mathcal{A}$ and the transition relation of a process skeleton $\mathsf{Sk}(\mathcal{A})$: If there is a path p and a valuation v with $v \models p$, then v defines a single transition (s, t) of a process skeleton $\mathsf{Sk}(\mathcal{A})$, if for each variable $x \in \Lambda \cup \Gamma \cup \{sv\}$ it holds that $s.x = v(x)$ and $t.x = v(x')$ and for each parameter variable $z \in \Pi$, $s.z = t.z = v(z)$. Finally, the initial states S^0 need to be specified. For the type of algorithms we consider in this paper, all variables of the skeleton that range over D are initialized to 0, and *sv* ranging over *SV* takes an initial value from a fixed subset of *SV*. (For other algorithms, or self-stabilizing systems, one would choose different initializations.)

Remark 3: It might seem restrictive that our guards do not contain, e.g., increment, assignments, non-deterministic choice from a range of values. However, all these statements can be translated in our form using the SSA transformation algorithm from [11]. For instance, Figure 1 has been obtained from the Promela case study in [24], which contains the mentioned statements. Figures 1 and 2 provide two of the algorithms we have used for our experiments in Section VII.

Definition 4 (PMCP for CFA): We define the Parameterized Model Checking Problem for CFA $\mathcal{A}$ by specializing Definition 2 to the parameterized process skeleton $\mathsf{Sk}(\mathcal{A})$.

The problem given in Definition 4 is undecidable even if the CFA contains only status variables [23].

V. Abstraction Scheme

The input to our abstraction method is the infinite parameterized family $\mathcal{F} = \{\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \mid \mathbf{p} \in \mathbf{P}_{RC}\}$ of Kripke structures specified via a CFA $\mathcal{A}$. The family $\mathcal{F}$ has two principal sources of unboundedness: unbounded variables in the process skeleton $\mathsf{Sk}(\mathcal{A})$, and the unbounded number of processes $N(\mathbf{p})$. We deal with these two aspects separately, using two abstraction steps, namely the *PIA data abstraction* and the *PIA counter abstraction*. In both abstraction steps we use the parametric interval abstraction PIA.

Given a CFA $\mathcal{A}$, let $\mathcal{G}_\mathcal{A}$ be the set of all linear combinations $a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i$ in the left-hand sides of $\mathcal{A}$'s threshold guards. Every expression ε of $\mathcal{G}_\mathcal{A}$ defines a function $f_\varepsilon \colon \mathbf{P}_{RC} \to D$. Let $\mathcal{T} = \{0, 1\} \cup \{f_\varepsilon \mid \varepsilon \in \mathcal{G}_\mathcal{A}\}$ be a finite *threshold set*, and $\mu + 1$ its cardinality. For convenience, we name elements of $\mathcal{T}$ as $\theta_0, \theta_1, \ldots, \theta_\mu$ with θ_0 corresponding to the constant function 0, and θ_1 corresponding to the constant 1. E.g., the CFA in Fig. 1 has the threshold set $\{\theta_0, \theta_1, \theta_2, \theta_3\}$, where $\theta_2(n, t, f) = t + 1$ and $\theta_3(n, t, f) = n - t$. Then, we define the domain of parametric intervals as:

$$\widehat{D} = \{I_j \mid 0 \leq j \leq \mu\}$$

Our abstraction rests on an implicit property of many FTDAs, namely, that the resilience condition *RC* induces an order on the thresholds used in the algorithm (e.g., $t+1 < n-t$).

Definition 5: The finite set $\mathcal{T}$ is uniformly ordered if for all $\mathbf{p} \in \mathbf{P}_{RC}$, and all $\theta_j(\mathbf{p})$ and $\theta_k(\mathbf{p})$ in $\mathcal{T}$ with $0 \leq j < k \leq \mu$, it holds that $\theta_j(\mathbf{p}) < \theta_k(\mathbf{p})$.

Assuming such an order does not limit the application of our approach: In cases where only a partial order is induced by *RC*, one can simply enumerate all finitely many total orders. As parameters, and thus thresholds, are kept unchanged in a run, one can verify an algorithm for each threshold order separately, and then combine the results.

Definition 5 allows us to properly define the *parameterized abstraction function* $\alpha_\mathbf{p} \colon D \to \widehat{D}$ and the *parameterized concretization function* $\gamma_\mathbf{p} \colon \widehat{D} \to 2^D$.

$$\alpha_\mathbf{p}(x) = \begin{cases} I_j & \text{if } x \in [\theta_j(\mathbf{p}), \theta_{j+1}(\mathbf{p})[\text{ for some } 0 \leq j < \mu \\ I_\mu & \text{otherwise.} \end{cases}$$

$$\gamma_\mathbf{p}(I_j) = \begin{cases} [\theta_j(\mathbf{p}), \theta_{j+1}(\mathbf{p})[& \text{if } j < \mu \\ [\theta_\mu(\mathbf{p}), \infty[& \text{otherwise.} \end{cases}$$

From $\theta_0(\mathbf{p}) = 0$ and $\theta_1(\mathbf{p}) = 1$, it immediately follows that for all $\mathbf{p} \in \mathbf{P}_{RC}$, we have $\alpha_\mathbf{p}(0) = I_0$, $\alpha_\mathbf{p}(1) = I_1$, and $\gamma_\mathbf{p}(I_0) = \{0\}$. Moreover, from the definitions of α, γ, and Definition 5 one immediately obtains:

Proposition 6: For all $\mathbf{p}$ in $\mathbf{P}_{RC}$, and for all a in D, it holds that $a \in \gamma_\mathbf{p}(\alpha_\mathbf{p}(a))$.

Definition 7: We define comparison between parametric intervals I_k and I_ℓ as $I_k \leq I_\ell$ iff $k \leq \ell$.

The PIA domain has similarities to predicate abstraction since the interval borders are naturally expressed as predicates, and computations over PIA are directly reduced to SMT solvers. However, notions such as the order of Definition 7 are not naturally expressed in terms of predicate abstraction.

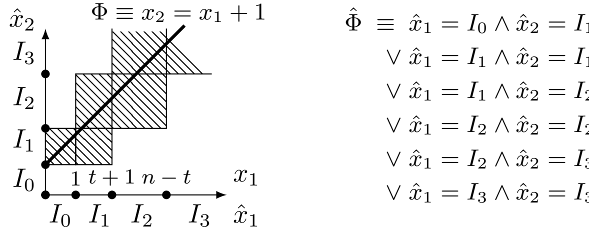


Fig. 4. The shaded area approximates the line $x_2 = x_1 + 1$ along the boundaries of our parametric intervals. Each shaded rectangle corresponds to one conjunctive clause in the formula to the right. Thus, given $\Phi \equiv x_2 = x_1 + 1$, the shaded rectangles correspond to $||\Phi||_\exists$, from which we immediately construct the existential abstraction $\hat{\Phi}$.

A. PIA data abstraction

We now discuss an existential abstraction of a formula Φ that is either a threshold or a comparison guard (we consider other guards later). To this end, we introduce notation for sets of vectors satisfying Φ. According to Section IV, formula Φ has two kinds of free variables: parameter variables from Π and data variables from $\Lambda \cup \Gamma$. Let $\mathbf{x}^p$ be a vector of parameter variables $(x_1^p, \ldots, x_{|\Pi|}^p)$ and $\mathbf{x}^v$ be a vector of variables $(x_1^v, \ldots, x_k^v)$ over D^k. Given a k-dimensional vector $\mathbf{d}$ of values from D, by

$$\mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi$$

we denote that Φ is satisfied on concrete values $x_1^v = d_1, \ldots, x_k^v = d_k$ and parameter values $\mathbf{p}$. Then, we define:

$$||\Phi||_\exists = \{\hat{\mathbf{d}} \in \widehat{D}^k \mid \exists \mathbf{p} \in \mathbf{P}_{RC}\ \exists \mathbf{d} = (d_1, \ldots, d_k) \in D^k. \\ \hat{\mathbf{d}} = (\alpha_{\mathbf{p}}(d_1), \ldots, \alpha_{\mathbf{p}}(d_k)) \ \wedge\ \mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi\}$$

Hence, the set $||\Phi||_\exists$ contains all vectors of abstract values that correspond to some concrete values satisfying Φ. Parameters do not appear anymore due to existential quantification. A PIA *existential abstraction* of Φ is defined to be a formula $\hat{\Phi}$ over a vector of variables $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_k)$ over $\widehat{D}^k$ such that $\{\hat{\mathbf{d}} \in \widehat{D}^k \mid \hat{\mathbf{x}} = \hat{\mathbf{d}} \models \hat{\Phi}\} \supseteq ||\Phi||_\exists$.

Computing PIA abstractions. The central property of our abstract domain is that it allows to abstract comparisons against thresholds (i.e., threshold guards) in a precise way. That is, we can abstract formulas of the form $\theta_j(\mathbf{p}) \le x_1$ by $I_j \le \hat{x}_1$ and $\theta_j(\mathbf{p}) > x_1$ by $I_j > \hat{x}_1$. In fact, this abstraction is precise in the following sense.

Proposition 8: For all $\mathbf{p} \in \mathbf{P}_{RC}$ and all $a \in D$: $\theta_j(\mathbf{p}) \le a$ iff $I_j \le \alpha_{\mathbf{p}}(a)$, and $\theta_j(\mathbf{p}) > a$ iff $I_j > \alpha_{\mathbf{p}}(a)$.

For comparison guards we use the general form, well-known from the literature, from the following proposition.

Proposition 9: If Φ is a formula over variables $x_1, \ldots, x_k$ over D, then $\bigvee_{(\hat{d}_1, \ldots, \hat{d}_k) \in ||\Phi||_\exists} \hat{x}_1 = \hat{d}_1 \wedge \cdots \wedge \hat{x}_k = \hat{d}_k$ is a PIA existential abstraction.

If the domain $\widehat{D}$ is small (as it is in our case), then one can enumerate all vectors of abstract values in $\widehat{D}^k$ and check which belong to our abstraction $||\Phi||_\exists$, using an SMT solver. As example consider the PIA domain $\{I_0, I_1, I_2, I_3\}$ for the CFA from Fig. 1. Fig. 4 illustrates $||\Phi||_\exists$ of $x_2 = x_1 + 1$ and the use of the formula from Proposition 9.

Transforming CFA. We now describe a general method to abstract `guard` formulas, and thus construct an abstract process skeleton. To this end, we denote by α_E a mapping from a concrete formula Φ to some existential abstraction of Φ (not necessarily constructed as above). By fixing α_E, we can define an abstraction of a `guard` of a CFA:

$$abs(g) = \begin{cases} \alpha_E(g) & \text{if } g \text{ is a threshold guard} \\ \alpha_E(g) & \text{if } g \text{ is a comparison guard} \\ g & \text{if } g \text{ is a status guard} \\ abs(g_1) \wedge abs(g_2) & \text{otherwise, i.e., } g \text{ is } g_1 \wedge g_2 \end{cases}$$

By abusing the notation, for a CFA $\mathcal{A}$ by $abs(\mathcal{A})$ we denote the CFA that is obtained from $\mathcal{A}$ by replacing every guard g with $abs(g)$. Note that $abs(\mathcal{A})$ contains only guards over sv and over abstract variables over $\widehat{D}$. For model checking, we have to reason about the Kripke structures that are built using the skeletons obtained from CFAs. We denote by $\mathsf{Sk}_{abs}(\mathcal{A})$, the process skeleton that is induced by CFA $abs(\mathcal{A})$, and by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$ an instance constructed from $\mathsf{Sk}_{abs}(\mathcal{A})$.

Soundness. It can be shown that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all CFA $\mathcal{A}$, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is simulated by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$, with respect to AP_{SV}. Moreover, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$.

B. PIA counter abstraction

In this section, we present a counter abstraction inspired by [29], which maps a system instance composed of *identical finite state* process skeletons to a single finite state system. We use the PIA domain $\widehat{D}$ along with abstractions $\alpha_E(\{x' = x + 1\})$ and $\alpha_E(\{x' = x - 1\})$ for the counters.

Let us consider a process skeleton $\mathsf{Sk} = (S, S_0, R)$, where $S = SV \times \tilde{D}^{|\Lambda|} \times \tilde{D}^{|\Gamma|} \times \tilde{D}^{|\Pi|}$ that is defined using an arbitrary finite domain $\tilde{D}$. We present counter abstraction over the abstract domain $\widehat{D}$ in two stages, where the first stage is only a change in representation, but not an abstraction.

Stage 1: Vector Addition System with States (VASS). Let $L = \{\ell \in SV \times \tilde{D}^{|\Lambda|} \mid \exists s \in S.\ \ell =_{\{sv\} \cup \Lambda} s\}$ be the set of *local states* of a process skeleton. As the domain $\tilde{D}$ and the set of local variables Λ are finite, L is finite. We write the elements of L as $\ell_1, \ldots, \ell_{|L|}$. We define the counting function $K \colon S_I \times L \to D$ such that $K[\sigma, \ell]$ is the number of processes i whose local state is ℓ in global state $\sigma \in S_I$, i.e., $\sigma[i] =_{\{sv\} \cup \Lambda} \ell$. Thus, we represent the system state σ as a tuple $(g_1, \ldots, g_k, K[\sigma, \ell_1], \ldots, K[\sigma, \ell_{|L|}])$, i.e., by the shared global state and by the counters for the local states. If a process moves from local state ℓ_i to local state ℓ_j, the counters of ℓ_i and ℓ_j will decrement and increment, respectively.

Stage 2: Abstraction of VASS. We abstract the counters K of the VASS representation using the PIA domain to obtain a finite state Kripke structure $\mathsf{C}(\mathsf{Sk})$. To compute $\mathsf{C}(\mathsf{Sk}) = (S_{\mathsf{C}}, S_{\mathsf{C}}^0, R_{\mathsf{C}}, \mathrm{AP}, \lambda_{\mathsf{C}})$ we proceed as follows:

A state $w \in S_{\mathsf{C}}$ is given by values of shared variables from the set Γ, ranging over $\tilde{D}^{|\Gamma|}$, and by a vector

$(\kappa[\ell_1], \ldots, \kappa[\ell_{|L|}])$ over the abstract domain $\widehat{D}$ from Section V. More concisely, $S_{\mathsf{C}} = \widehat{D}^{|L|} \times \tilde{D}^{|\Gamma|}$.

Definition 10: The parameterized abstraction mapping $\bar{h}^{cnt}_{\mathbf{p}}$ maps a global state σ of the system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ to a state w of the abstraction $\mathsf{C}(\mathsf{Sk})$ such that: For all $\ell \in L$ it holds that $w.\kappa[\ell] = \alpha_{\mathbf{p}}(K[\sigma, \ell])$, and $w =_{\Gamma} \sigma$.

From the definition, one can see how to construct the initial states. Informally, we require (1) that the initial shared states of $\mathsf{C}(\mathsf{Sk})$ correspond to initial shared states of Sk, (2) that there are actually $N(\mathbf{p})$ processes in the system, and (3) that initially all processes are in an initial state.

The intuition for the construction of the transition relation is as follows: Like in VASS, a step that brings a process from local state ℓ_i to ℓ_j can be modeled by decrementing the (non-zero) counter of ℓ_i and incrementing the counter of ℓ_j using the existential abstraction $\alpha_E(\{\kappa'[\ell_i] = \kappa[\ell_i] - 1\})$ and $\alpha_E(\{\kappa'[\ell_j] = \kappa[\ell_j] + 1\})$.

Soundness. We show that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all finite state process skeletons Sk, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is simulated by $\mathsf{C}(\mathsf{Sk})$, w.r.t. AP_{SV}. Further, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a J-fair path of $\mathsf{C}(\mathsf{Sk})$.

Theorem 11 (Soundness of data & counter abstraction): For all CFA $\mathcal{A}$, and for all formulas φ from $\mathsf{LTL}_{\text{-}\mathsf{X}}$ over AP_{SV} and justice constraints $J \subseteq \mathrm{AP}_D$: if $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A})) \models_J \varphi$, then for all $\mathbf{p} \in \mathbf{P}_{RC}$ it holds $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \models_J \varphi$.

VI. Abstraction Refinement

The states of the abstract system are determined by variables over $\widehat{D}$. Proposition 8 shows that we precisely abstract the relevant properties of our variables, i.e., comparisons to thresholds. Hence, the classic CEGAR approach [7], which consists of refining the state space, does not appear suitable. However, the non-determinism due to our existential abstraction leads to *spurious transitions* that one can eliminate.

We encountered two sources of spurious transitions: As discussed in Section II, transitions can "lose processes," i.e., any concretization of the abstract number of processes is less than the number of processes we started with. This is not within the assumption of FTDAs, and thus spurious. Second, in our case study (cf. Figure 1) processes increase the global variable *nsnt* by one, when they transfer to a state where the value of the status variable is in $\{\mathrm{SE}, \mathrm{AC}\}$. Hence, in concrete system instances, *nsnt* should always be equal to the number of processes whose status variable value is in $\{\mathrm{SE}, \mathrm{AC}\}$, while due to phenomena similar to those discussed above, we can "lose messages" in the abstract system.

The experiments show that in our case studies neither losing processes nor losing messages has influence on the verification of safety specifications. However, these behaviors pose challenges for liveness as they lead to spurious counterexamples: Message passing FTDAs typically require that a process receives messages from (nearly) all correct processes, which is problematic if processes (i.e., potential senders) or messages are lost.

Besides, in Figure 1 we model message receptions by an update of the variable *rcvd*, more precisely, $rcvd \le rcvd' \wedge rcvd' \le nsnt + f$. One may observe that this alone does not require that the value of *rcvd* actually increases. Hence, we add justice requirements, e.g., $J = \{[\forall i.\ rcvd_i \ge nsnt]\}$ in our case study. As observed by [29], counter abstraction may lead to justice suppression. Given a counter-example in the form of a lasso, we detect whether its loop contains only unjust states. If this is the case, similar to an idea from [29], we refine $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by adding a justice requirement, which is consistent with existing requirements in all concrete instances.

Below, we give a general framework for a sound refinement of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$. (In [23], we provide a more detailed discussion on the practical refinement techniques that we use in our experiments.) To simplify presentation, we define a *monster system* as a (possibly infinite) Kripke structure $\mathsf{Sys}_\omega = (S_\omega, S^0_\omega, R_\omega, \mathrm{AP}, \lambda_\omega)$, whose state space and transition relation are disjoint unions of state spaces and transition relations of system instances $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) = (S_{\mathbf{p}}, S^0_{\mathbf{p}}, R_{\mathbf{p}}, \mathrm{AP}, \lambda_{\mathbf{p}})$ over all admissible parameters:

$$S_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} S_{\mathbf{p}}, \qquad S^0_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} S^0_{\mathbf{p}}, \qquad R_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} R_{\mathbf{p}}$$
$$\lambda_\omega : S_\omega \to 2^{\mathrm{AP}} \text{ and } \forall \mathbf{p} \in \mathbf{P}_{RC}, \forall s \in S_{\mathbf{p}}.\ \lambda_\omega(s) = \lambda_{\mathbf{p}}(s)$$

Let $h\colon S_\omega \to S_{\mathsf{C}}$ be an abstraction mapping, e.g., a combination of the abstraction mappings from Section V.

Definition 12: A sequence $T = \{\sigma_i\}_{i \ge 1}$ is a *concretization* of path $\hat{T} = \{w_i\}_{i \ge 1}$ from $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ if and only if $\sigma_1 \in S^0_\omega$ and for all $i \ge 1$ it holds $h(\sigma_i) = w_i$.

Definition 13: A path $\hat{T}$ of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ is a *spurious path* iff every concretization T of $\hat{T}$ is not a path in Sys_ω.

A prerequisite to abstraction refinement is to check whether a counter-example provided by the model checker is spurious. While for finite state systems there are methods to detect whether a path is spurious [7], we are not aware of a method to detect whether a path $\hat{T}$ in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ corresponds to a path in the (concrete) infinite monster system Sys_ω. Therefore, we limit ourselves to detecting and refining uniformly spurious transitions and unjust states. We first consider spurious transitions.

Definition 14: An abstract transition $(w, w') \in R_{\mathsf{C}}$ is *uniformly spurious* iff there is no transition $(\sigma, \sigma') \in R_\omega$ with $w = h(\sigma)$ and $w' = h(\sigma')$.

The following theorem provides us with a general criterion that ensures that removing uniformly spurious transitions does not affect the property of transition preservation.

Theorem 15: Let $T \subseteq R_{\mathsf{C}}$ be a set of spurious transitions. Then for every transition $(\sigma, \sigma') \in R_\omega$ there is a transition $(h(\sigma), h(\sigma'))$ in $R_{\mathsf{C}} \setminus T$.

It follows that the system $(S_{\mathsf{C}}, S^0_{\mathsf{C}}, R_{\mathsf{C}} \setminus T, \mathrm{AP}, \lambda_{\mathsf{C}})$ still simulates Sys_ω. After considering spurious transitions, we now consider justice suppression.

Definition 16: An abstract state $w \in S_{\mathsf{C}}$ is *unjust under* $q \in \mathrm{AP}_D$ iff there is no concrete state $\sigma \in S_\omega$ with $w = h(\sigma)$ and $q \in \lambda_\omega(\sigma)$.

Consider infinite counterexamples of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$, which have a form of lassos $w_1 \ldots w_k (w_{k+1} \ldots w_m)^\omega$. For such a counterexample $\hat{T}$ we denote the set of states in the lasso's loop by U. We then check, whether all states of U are unjust

under some justice constraint $q \in J$. If this is the case, then $\hat{T}$ is a spurious counterexample, because the justice constraint q is violated. Note that it is sound to only consider infinite paths, where states outside of U appear infinitely often; in fact, this is a justice requirement. To refine C's unjust behavior we add a corresponding justice requirement. Formally, we augment J (and AP_D) with a propositional symbol $[\mathit{off}\ U]$. Further, we augment the labelling function λ_{C} such that every $w \in S_{\mathsf{C}}$ is labelled with $[\mathit{off}\ U]$ if and only if $w \notin U$.

Theorem 17: Let $J \subseteq \text{AP}_D$ be a set of justice requirements, $q \in J$, and $U \subseteq S_{\mathsf{C}}$ be a set of unjust states under q. Let $\pi = \{\sigma_i\}_{i \geq 1}$ be an arbitrary fair path of Sys_ω under J. The path $\hat{\pi} = \{\hat{h}(\sigma_i)\}_{i \geq 1}$ is fair in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ under $J \cup \{[\mathit{off}\ U]\}$.

From this we derive that loops containing only unjust states can be eliminated, and thus $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ be refined.

We encountered cases where several non-uniform spurious transitions resulted in a uniformly spurious path (i.e., a counterexample). We refine such spurious behavior by invariants. These invariants are provided by the user as invariant candidates, and are then automatically checked to actually be invariants using an SMT solver. In our example the invariant is simply "the number of processes that sent a message equals the number of sent messages."

VII. Experimental Evaluation

To show feasibility of our abstractions, we have implemented the PIA abstractions and the refinement loop in OCaml as a prototype tool ByMC. We evaluated it on different broadcasting algorithms. They deal with different fault models and resilience conditions; the algorithms are: (Byz), which is the algorithm from Figure 1, for t Byzantine faults if $n > 3t$, (Symm) for t symmetric (identical Byzantine [2]) faults if $n > 2t$, (Omit) for t send omission faults if $n > 2t$, and (Clean) for t clean crash faults [37] if $n > t$. In addition, we verified the RBC algorithm — formalized also in [18] — whose CFA is given in Figure 2. In this paper we verify the following safety and liveness specifications:

$$[\forall i.\ sv_i \neq \text{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \text{AC}] \qquad \text{(U)}$$

$$[\forall i.\ sv_i = \text{V1}] \rightarrow \mathbf{F}\ [\exists j.\ sv_j = \text{AC}] \qquad \text{(C)}$$

$$\mathbf{G}\,(\neg\,[\exists i.\ sv_i = \text{AC}]) \vee \mathbf{F}\ [\forall j.\ sv_j = \text{AC}] \qquad \text{(R)}$$

In addition, in [18] a specification A for RBC was introduced, which we verify for RBC. In contrast to [18], we actually implemented our verification method and give experimental data.

From the literature we know that we cannot expect to verify these FTDAs without restricting the environment, e.g., with communication fairness, namely, every message sent is eventually received. To capture this, we use justice requirements, e.g., $J = \{[\forall i.\ \mathit{rcvd}_i \geq \mathit{nsnt}]\}$ in the Byzantine case.

We extended Promela [22] with constructs to express Π, AP, *RC*, and N [24]. ByMC receives a description of a CFA $\mathcal{A}$ in this extended Promela, and then syntactically extracts the thresholds. The tool chain uses the Yices SMT solver for existential abstraction, and generates the counter abstraction $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ in standard Promela, such that we can use Spin to do finite state model checking. Finally, ByMC also implements the refinements introduced in Section VI and refines the Promela code for $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by introducing predicates capturing spurious transitions and unjust states.

TABLE I. Summary of experiments

$M \models \varphi$?	RC	Spin Time	Spin Memory	Spin States	Spin Depth	$\lvert\widehat{D}\rvert$	#R	Total Time
$Byz \models U$	(A)	2.3 s	82 MB	483k	9154	4	0	4 s
$Byz \models C$	(A)	3.5 s	104 MB	970k	20626	4	10	32 s
$Byz \models R$	(A)	6.3 s	107 MB	1327k	20844	4	10	24 s
$Sym \models U$	(A)	0.1 s	67 MB	19k	897	3	0	1 s
$Sym \models C$	(A)	0.1 s	67 MB	19k	1113	3	2	3 s
$Sym \models R$	(A)	0.3 s	69 MB	87k	2047	3	12	16 s
$Omt \models U$	(A)	0.1 s	66 MB	4k	487	3	0	1 s
$Omt \models C$	(A)	0.1 s	66 MB	7k	747	3	5	6 s
$Omt \models R$	(A)	0.1 s	66 MB	8k	704	3	5	10 s
$Cln \models U$	(A)	0.3 s	67 MB	30k	1371	3	0	2 s
$Cln \models C$	(A)	0.4 s	67 MB	35k	1707	3	4	8 s
$Cln \models R$	(A)	1.1 s	67 MB	51k	2162	3	13	31 s
$RBC \models U$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
$RBC \models A$	—	0.1 s	66 MB	1.7k	333	2	0	1 s
$RBC \models R$	—	0.1 s	66 MB	1.2k	259	2	0	1 s
$RBC \not\models C$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
$Byz \not\models U$	(B)	5.2 s	101 MB	1093k	17685	4	9	56 s
$Byz \not\models C$	(B)	3.7 s	102 MB	980k	19772	4	11	52 s
$Byz \not\models R$	(B)	0.4 s	67 MB	59k	6194	4	10	17 s
$Byz \models U$	(C)	3.4 s	87 MB	655k	10385	4	0	5 s
$Byz \models C$	(C)	3.9 s	101 MB	963k	20651	4	9	32 s
$Byz \not\models R$	(C)	2.1 s	91 MB	797k	14172	4	30	78 s
$Sym \not\models U$	(B)	0.1 s	67 MB	19k	947	3	0	2 s
$Sym \not\models C$	(B)	0.1 s	67 MB	18k	1175	3	2	4 s
$Sym \models R$	(B)	0.2 s	67 MB	42k	1681	3	8	12 s
$Omt \models U$	(D)	0.1 s	66 MB	5k	487	3	0	1 s
$Omt \not\models C$	(D)	0.1 s	66 MB	5k	487	3	0	2 s
$Omt \not\models R$	(D)	0.1 s	66 MB	0.1k	401	3	0	2 s

Table I summarizes our experiments run on 3.3GHz Intel® Core™ 4GB. In the cases (A) we used resilience conditions as provided by the literature, and verified the specification. The model RBC is the reliable broadcast algorithm also considered in [18] under the resilience condition $n \geq t \geq f$. In the bottom part of Table I we used different resilience conditions under which we expected the algorithms to fail. The cases (B) capture the case where more faults occur than expected by the algorithm designer ($f \leq t + 1$ instead of $f \leq t$), while the cases (C) and (D) capture the cases where the algorithms were designed by assuming wrong resilience conditions (e.g., $n \geq 3t$ instead of $n > 3t$ in the Byzantine case). We omit (Clean) as the only sensible case $n = t = f$ (all processes are faulty) results into a trivial abstract domain of one interval $[0, \infty)$. The column "#R" gives the numbers of refinement steps. In the cases where it is greater than zero, refinement was necessary, and "Spin Time" refers to the Spin running time after the last refinement step. Finally, column $|\widehat{D}|$ indicates the size of the abstract domain.

VIII. Related work

Traditionally, correctness of FTDAs is shown by handwritten proofs [27], [2], and, in some cases, by proof assistants [26], [31], [5]. Completely automated model checking or synthesis are usually not parameterized [24], [36], [34], [3]. Our work stands in the tradition of parameterized model checking for protocols [4], [20], [17], [29], [8], e.g., mutual exclusion and cache coherence. In particular, counter abstraction and justice preservation by Pnueli et al. [29] are keystones of our work.

To the best of our knowledge there are two papers on parameterized model checking of FTDAs [18], [1]. The authors of [18] use regular model checking to make interesting theoretical progress, but did not do any implementation. Their models are limited to processes whose local state space and transition relation are *finite and independent of parameters*. This was sufficient to formalize a reliable broadcast algorithm that tolerates crash faults, and where every process stores whether it has received at least one message. Such models are *not sufficient* to capture FTDAs that contain threshold guards as in our case. Moreover, the presence of a resilience condition such as $n > 3t$ would require them to intersect the regular languages, which describe sets of states, with context-free languages that enforce the resilience condition.

In [1], the safety of synchronous broadcasting algorithms that tolerate crash or send omission faults has been verified. These FTDAs have similar restrictions as the ones considered in [18]: Alberti et al. [1] mention that they did not consider FTDAs that feature “substantial arithmetic reasoning”, i.e., threshold guards and resilience conditions, as they would require novel suitable techniques. Our abstractions address this arithmetic reasoning.

To the best of our knowledge, the current paper is thus the first in which *safety* and *liveness* of an FTDA that tolerates Byzantine faults has been *automatically* verified for *all* system sizes and *all* admissible numbers of faulty processes.

IX. Conclusions

We extended the standard setting of parameterized model checking to processes that use threshold guards, and are parameterized with a resilience condition. As a case study we have chosen the core of several broadcasting algorithms under different failure models, including one [33] that tolerates Byzantine faults. These algorithms are widely applied in the literature: typically, multiple (possibly an unbounded number of) instances are used in combination. As future work, we plan to use compositional model checking techniques [28] for parameterized verification of such algorithms. Another open issue is to capture additional fault assumptions such as communication faults [5], [37].

References

[1] 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,” *JSAT*, vol. 8, no. 1/2, pp. 29–61, 2012.

[2] H. Attiya and J. Welch, *Distributed Computing*, 2nd ed. Wiley, 2004.

[3] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad, “Symbolic synthesis of masking fault-tolerant distributed programs,” *Distributed Computing*, vol. 25, no. 1, pp. 83–108, 2012.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg, “Reasoning about networks with many identical finite state processes,” *Inf. Comput.*, vol. 81, pp. 13–31, 1989.

[5] B. Charron-Bost and S. Merz, “Formal verification of a consensus algorithm in the heard-of model,” *IJSI*, vol. 3, no. 2–3, pp. 273–303, 2009.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[8] E. Clarke, M. Talupur, and H. Veith, “Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems,” in *TACAS’08/ETAPS’08*. Springer, 2008, pp. 33–47.

[9] E. M. Clarke, O. Grumberg, and D. E. Long, “Model checking and abstraction,” *ACM TOPLAS*, vol. 16, no. 5, pp. 1512–1542, 1994.

[10] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *POPL*. ACM, 1977, pp. 238–252.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.

[12] D. Dams, R. Gerth, and O. Grumberg, “Abstract interpretation of reactive systems,” *ACM TOPLAS*, vol. 19, no. 2, pp. 253–291, 1997.

[13] R. De Prisco, D. Malkhi, and M. K. Reiter, “On k-set consensus problems in asynchronous systems,” *TPDS*, vol. 12, no. 1, pp. 7–21, 2001.

[14] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, “Reaching approximate agreement in the presence of faults,” *J. ACM*, vol. 33, no. 3, pp. 499–516, 1986.

[15] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *J.ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[16] E. A. Emerson and V. Kahlon, “Reducing model checking of the many to the few,” in *CADE*, ser. LNCS, 2000, vol. 1831, pp. 236–254.

[17] ——, “Exact and efficient verification of parameterized cache coherence protocols,” in *CHARME*, ser. LNCS, vol. 2860, 2003, pp. 247–262.

[18] D. Fisman, O. Kupferman, and Y. Lustig, “On verifying fault tolerance of distributed protocols,” in *TACAS*, ser. LNCS, vol. 4963, 2008, pp. 315–331.

[19] M. Függer and U. Schmid, “Reconciling fault-tolerant distributed computing and systems-on-chip,” *Dist. Comp.*, vol. 24, no. 6, pp. 323–355, 2012.

[20] S. M. German and A. P. Sistla, “Reasoning about systems with many processes,” *J. ACM*, vol. 39, pp. 675–735, 1992.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *POPL*. ACM, 2002, pp. 58–70.

[22] G. Holzmann, *The SPIN Model Checker*. Addison-Wesley, 2003.

[23] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder, “Counter attack on Byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms,” *arXiv CoRR*, vol. abs/1210.3846, 2012.

[24] ——, “Towards modeling and model checking fault-tolerant distributed algorithms,” in *SPIN*, ser. LNCS, vol. 7976, 2013, pp. 209–226.

[25] Y. Kesten and A. Pnueli, “Control and data abstraction: the cornerstones of practical formal verification,” *STTT*, vol. 2, pp. 328–342, 2000.

[26] P. Lincoln and J. Rushby, “A formally verified algorithm for interactive consistency under a hybrid fault model,” in *FTCS*, 1993, pp. 402–411.

[27] N. Lynch, *Distributed Algorithms*. Morgan Kaufman, 1996.

[28] K. L. McMillan, “Parameterized verification of the flash cache coherence protocol by compositional model checking,” in *CHARME*, ser. LNCS, vol. 2144, 2001, pp. 179–195.

[29] A. Pnueli, J. Xu, and L. Zuck, “Liveness with $(0,1,\infty)$- counter abstraction,” in *CAV*, ser. LNCS. Springer, 2002, vol. 2404, pp. 93–111.

[30] S. Sankaranarayanan, F. Ivancic, and A. Gupta, “Program analysis using symbolic ranges,” in *SAS*, ser. LNCS, vol. 4634, 2007, pp. 366–383.

[31] U. Schmid, B. Weiss, and J. Rushby, “Formally verified Byzantine agreement in presence of link faults,” in *ICDCS*, 2002, pp. 608–616.

[32] T. K. Srikanth and S. Toueg, “Optimal clock synchronization,” *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, 1987.

[33] T. Srikanth and S. Toueg, “Simulating authenticated broadcasts to derive simple fault-tolerant algorithms,” *Dist. Comp.*, vol. 2, pp. 80–94, 1987.

[34] W. Steiner, J. M. Rushby, M. Sorea, and H. Pfeifer, “Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation,” in *DSN*, 2004, pp. 189–198.

[35] M. Talupur and M. R. Tuttle, “Going with the flow: Parameterized verification using message flows,” in *FMCAD*, 2008, pp. 1–8.

[36] T. Tsuchiya and A. Schiper, “Verification of consensus algorithms using satisfiability solving,” *Dist. Comp.*, vol. 23, no. 5–6, pp. 341–358, 2011.

[37] J. Widder and U. Schmid, “Booting clock synchronization in partially synchronous systems with hybrid process and link failures,” *Dist. Comp.*, vol. 20, no. 2, pp. 115–140, 2007.

Part II

Parameterized and Bounded Model Checking of Threshold-Guarded Distributed Algorithms with SMT

Chapter 4

On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability

Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. Journal of Information and Computation, vol. 252, pp. 95-109, 2017.

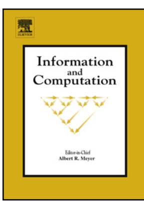
DOI: http://dx.doi.org/10.1016/j.ic.2016.03.006

Information and Computation 252 (2017) 95–109

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco



On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability ☆

Igor Konnov, Helmut Veith, Josef Widder *

TU Wien (Vienna University of Technology), Austria

A R T I C L E I N F O

Article history:
Received 15 December 2014
Available online 2 March 2016

Keywords:
Model checking
Fault-tolerant distributed algorithms
Byzantine faults
Computational models

A B S T R A C T

Counter abstraction is a powerful tool for parameterized model checking, if the number of local states of the concurrent processes is relatively small. In recent work, we introduced parametric interval counter abstraction that allowed us to verify the safety and liveness of threshold-based fault-tolerant distributed algorithms (FTDA). Due to state space explosion, applying this technique to distributed algorithms with hundreds of local states is challenging for state-of-the-art model checkers. In this paper, we demonstrate that reachability properties of FTDAs can be verified by bounded model checking. To ensure completeness, we need an upper bound on the distance between states. We show that the diameters of accelerated counter systems of FTDAs, and of their counter abstractions, have a quadratic upper bound in the number of local transitions. Our experiments show that the resulting bounds are sufficiently small to use bounded model checking for parameterized verification of reachability properties of several FTDAs, some of which have not been automatically verified before.

© 2016 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

A system that consists of concurrent anonymous (identical) processes can be modeled as a counter system: Instead of recording which process is in which local state, we record for each local state, how many processes are in this state. We have one counter per local state ℓ, denoted by $\kappa[\ell]$. Each counter is bounded by the number of processes. A step by a process that goes from local state ℓ to local state ℓ' is modeled by decrementing $\kappa[\ell]$ and incrementing $\kappa[\ell']$.

We consider a specific class of counter systems, namely those that are defined by *threshold automata*. The technical motivation to introduce threshold automata is to capture the relevant properties of fault-tolerant distributed algorithms (FTDAs). FTDAs are an important class of distributed algorithms that work even if a subset of the processes fails [26]. Typically, they are parameterized in the number of processes and the number of tolerated faulty processes. These numbers of processes are parameters of the verification problem. We show that the counter systems defined by threshold automata have a diameter whose bound is independent of the bound on the counters, but depends only on characteristics of the threshold automaton. This bound can be used for parameterized model checking of FTDAs, as we confirm by experimental evaluation.

☆ 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) and grant PROSEED.

* Corresponding author.
E-mail address: widder@forsyte.at (J. Widder).

http://dx.doi.org/10.1016/j.ic.2016.03.006

0890-5401/© 2016 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Modeling FTDAs as counter systems defined by threshold automata A threshold automaton consists of rules that define the conditions and effects of changes to the local state of a process of a distributed algorithm. Conditions are *threshold guards* that compare the value of a shared variable to a linear combination of parameters, e.g., $x \geq n - t$, where x is a shared variable and n and t are parameters. This captures counting arguments which are used in FTDAs, e.g., a process takes a certain step only, if it has received a message from a majority of processes. To model this, we use the shared variable x as the number of processes that have sent a message, n as the number of processes in the system, and t as the assumed number of faulty processes. The condition $x \geq n - t$ then captures a majority under the resilience condition that $n > 2t$. Resilience conditions are standard assumptions for the correctness of an FTDA.[1] The effect of a rule of a threshold automaton is that a shared variable is increased, which naturally captures that a process has sent a message. As a process cannot undo the sending of a message, it is natural to consider threshold automata where shared variables are never decreased. In addition, we use shared variables to model the number of processes that have sent a specific message. To be able to do so, we have to restrict how often a process may send a specific message. In particular, to model the counting mechanism, we have to prevent that a process sends a message from within an infinite loop (or a loop where the number of iterations is unknown). We are thus led to consider threshold automata where rules that form cycles do not modify shared variables. While we add this restriction to derive our technical contribution, we do not consider it too limiting with respect to the application domain: Indeed, in all our case studies a process sends a given message at most once; this property appears natural if one considers distributed algorithms under the classic assumption of reliable communication.

Bounding the diameter For reachability it is not relevant whether we "move" processes one by one from local state ℓ to local state ℓ'. If several processes perform the same transition one after the other, we can model this as a single update on the counters: The sequence where b processes one after the other move from ℓ to ℓ' can be encoded as a single transition where $\kappa[\ell]$ is decreased by b and $\kappa[\ell']$ is increased by b. We call the value of b the *acceleration factor*. It may vary in a run depending on how many repetitions of the same transition should be captured. We call such runs of a counter system *accelerated*. The lengths of accelerated runs are the ones relevant for the diameter of the counter system.

Our central idea is that given a run that starts in configuration σ and ends in configuration σ', by swapping and accelerating transitions in that run, we can construct a run of bounded length that also starts in σ and ends in σ'. This bound then gives us the diameter. For deriving this bound, the main technical challenge comes from the interactions of shared variables and threshold guards. We address it with the following three ideas:

i. *Acceleration*. As discussed above.
ii. *Sorting*. Given an arbitrary run of a counter system, we can shorten it by changing the order of transitions such that there are possibly many consecutive transitions that can be merged according to (i), and the resulting run leads to the same configuration as the original run. However, as we have arithmetic threshold conditions, not all changes of the order result in allowed runs.
iii. *Segmentation*. We partition a run into segments, inside of which we can reorder the transitions; cf. (ii).

In combination, these three ideas enable us to prove the main theorem: *The diameter of a counter system is at most quadratic in the number of rules; more precisely, it is bounded by the product of the number of rules and the number of distinct threshold conditions.* In particular, the diameter is independent of the parameter values.

Using the bound for parameterized model checking Parameterized model checking is concerned with the verification of concurrent or distributed systems, where the number of processes is not a priori fixed, that is, a system is verified for all sizes [6]. In our case, the counter systems for all values of n and t that satisfy the resilience condition should be verified. A well-known parameterized model checking technique is to map all these counter systems to a *counter abstraction*, where the counter values are not natural numbers, but range over an abstract finite domain [30]. In [14], we developed a more general form of counter abstraction for expressions used in threshold guards, which leads, e.g., to the abstract domain of four values that capture the parametric intervals $[0, 1)$ and $[1, t + 1)$ and $[t + 1, n - t)$ and $[n - t, \infty)$. It is easy to see [14] that a counter abstraction simulates all counter systems for all parameter values that satisfy the resilience condition. The bound d on the diameter of counter systems implies a bound $\hat{d}$ on the diameter of the counter abstraction. From this and simulation follows that if an abstract state is not reachable in the counter abstraction within $\hat{d}$ steps, then no concretization of this state is reachable in any of the concrete counter systems. This allows us to efficiently combine counter abstraction with *bounded model checking* [5]. Typically, bounded model checking is restricted to finding bugs that occur after a bounded number of steps of the systems. However, if one can show that within this bound every state is reachable from an initial state, bounded model checking is a complete method for verifying reachability.

[1] Indeed much research in distributed algorithms is devoted to show that certain problems are solvable only under some resilience condition, e.g., the seminal result on Byzantine fault tolerance by Pease et al. [28].

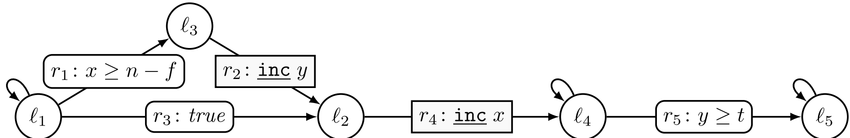


Fig. 1. Example of a threshold automaton.

2. Our approach at a glance

Fig. 1 represents a threshold automaton: The circles depict the local states, and the arrows represent rules (r_1 to r_5) that define how the automaton makes transitions. Rounded corner labels correspond to conditional rules, so that the rule can only be executed if the threshold guard evaluates to true. In our example, x and y are shared variables, and n, t, and f are parameters. We assume that they satisfy the resilience condition $n \geq 2t \wedge f \leq t$. The number of processes (that each execute the automaton) depends on the parameters, in this example we assume that n processes run concurrently. Finally, rectangular labels on arrows correspond to rules that increment a shared variable. The transitions of the counter system are then defined using the rules, e.g., when rule r_2 is executed, then variable y is incremented and the counters $\kappa[\ell_3]$ and $\kappa[\ell_2]$ are updated.

Consider a counter system in which the parameter values are $n = 3$, and $t = f = 1$. Let σ_0 be the configuration where $x = y = 0$ and all counters are 0 except $\kappa[\ell_1] = 3$. This configuration corresponds to a concurrent system where all three processes are in ℓ_1. For illustration, we assume that in this concurrent system processes have the identifiers 1, 2, and 3, and we denote by $r_i(j)$ that process j executes rule r_i. Recall that we have anonymous (symmetric) systems, so we use the identifiers only for illustration: the transition of the counter system is solely defined by the rule being executed.

As we are interested in the diameter, we have to consider the distance between configurations in terms of length of runs. In this example, we consider the distance of σ_0 to a configuration where $\kappa[\ell_5] = 3$, that is, all three processes are in local state ℓ_5. First, observe that the rule r_5 is locked in σ_0 as $y = 0$ and $t = 1$. Hence, we require that rule r_2 is executed at least once so that the value of y increases. However, due to the precedence relation on the rules, before that, r_1 must be executed, which is also locked in σ_0. The sequence of transitions $\tau_1 = r_3(1), r_4(1), r_3(2), r_4(2)$ leads from σ_0 to the configuration where $\kappa[\ell_1] = 1$, $\kappa[\ell_4] = 2$, and $x = 2$; we denote it by σ_1. In σ_1, rule r_1 is unlocked, so we may apply $\tau_2 = r_1(3), r_2(3)$, to arrive at σ_2, where $y = 1$, and thus r_5 is unlocked. To σ_2 we may apply $\tau_3 = r_5(1), r_5(2), r_4(3), r_5(3)$ to arrive at the required configuration σ_3 with $\kappa[\ell_5] = 3$.

In order to exploit acceleration as much as possible, we would like to group together occurrences of the same rule. In τ_1, we can actually swap $r_4(1)$ and $r_3(2)$ as locally the precedence relation of each process is maintained, and both rules are unconditional. Similarly, in τ_3, we can move $r_4(3)$ to the beginning of the sequence τ_3. Concatenating these altered sequences, the resulting schedule is $\tau = r_3(1), r_3(2), r_4(1), r_4(2), r_1(3), r_2(3), r_4(3), r_5(1), r_5(2), r_5(3)$. We can group together the consecutive occurrences for the same rules r_i, and write the schedule using pairs consisting of rules and acceleration factors, that is, $(r_3, 2)$, $(r_4, 2)$, $(r_1, 1)$, $(r_2, 1)$, $(r_4, 1)$, $(r_5, 3)$.

In schedule τ, the occurrences of all rules are grouped together except for r_4. That is, in the accelerated schedule we have two occurrences for r_4, while for the other rules one occurrence is sufficient. Actually, there is no way around this: We cannot swap $r_2(3)$ with $r_4(3)$, as we have to maintain the local precedence relation of process 3. More precisely, in the counter system, r_4 would require us to decrease the counter $\kappa[\ell_2]$ at a point in the schedule where $\kappa[\ell_2] = 0$. We first have to increase the counter value by executing a transition according to rule r_2, before we can apply r_4. Moreover, we cannot move the subsequence $r_1(3), r_2(3), r_4(3)$ to the left, as $r_1(3)$ is locked in the prefix.

In this paper we characterize such cases. The issue here is that r_4 can unlock r_1 (we use the notation $r_4 \prec_U r_1$), while r_1 precedes r_4 in the control flow of the processes ($r_1 \prec_P^+ r_4$). We coin the term *milestone* for transitions like $r_1(3)$ that cannot be moved, and show that the same issue arises if a rule r locks a threshold guard of rule r', where r precedes r' in the control flow. As processes do not decrease shared variables, we have at most one milestone per threshold guard. The sequence of transitions between milestones is called a segment. We prove that transitions inside a segment can be swapped, so that one can group transitions for the same rule in so-called batches. Each of these batches can then be replaced by a single accelerated transition that leads to the same configuration as the original batch. Hence, any segment can be replaced by an accelerated one whose length is at most the number of rules of a process. This, and the number of milestones, gives us the required bound on the diameter. This bound is independent of the parameters, and only depends on the number of threshold guards and the precedence relation between the rules of the processes.

Our main result is that the diameter is independent of the parameter values. In contrast, reachability of a specific local state depends on the parameter values: In order for a process to reach ℓ_5 in our example, at least $n - f$ processes must execute r_4 before at least t other processes must execute r_2. That is, the system must contain at least $(n - f) + t$ processes. In case of $t > f$, we obtain $(n - f) + t > n$, which is a contradiction, and ℓ_5 cannot be reached for such parameter values. The *model checking problem* we are interested in is whether *a given state is unreachable for all parameter values* that satisfy the resilience condition.

3. Parameterized counter systems

3.1. Threshold automata

A threshold automaton describes a process in a concurrent system. It is defined by its local states, shared variables, parameters, and by rules that define the state changes and their conditions and effects on shared variables. Formally, a *threshold automaton* is a tuple $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ defined below.

States The set $\mathcal{L}$ is the finite set of *local states*, and $\mathcal{I} \subseteq \mathcal{L}$ is the set of *initial local states*. (As we later will index counters by local states, for simplicity we use the convention that $\mathcal{L} = \{1, \dots, |\mathcal{L}|\}$.) The set Γ is the finite set of *shared variables* that range over $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. To simplify the presentation, we view the variables as vectors in $\mathbb{N}_0^{|\Gamma|}$. The finite set Π is a set of *parameter variables* that range over $\mathbb{N}_0$, and the *resilience condition RC* is a predicate over $\mathbb{N}_0^{|\Pi|}$. Then, we denote the set of *admissible parameters* by $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : RC(\mathbf{p})\}$.

Rules A rule defines a conditional transition between local states that may update the shared variables. The semantics is defined via counter systems in Section 3.2. Here we only give informal explanations of the semantics.

Formally, a *rule* is a tuple $(from, to, \varphi^{\le}, \varphi^{>}, \mathbf{u})$: The local states *from* and *to* are from $\mathcal{L}$. Intuitively, they capture from which local state to which a process moves, or, in terms of counter systems, which counters decrease and increase, respectively. A rule is only executed if the conditions $\varphi^{\le}$ and $\varphi^{>}$ evaluate to true. Each condition consists of multiple guards. Each guard is defined using some shared variable $x \in \Gamma$, and coefficients $a_0, \dots, a_{|\Pi|} \in \mathbb{Z}$, so that

$$a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i \le x \qquad \text{and} \qquad a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i > x$$

are a *lower guard* and *upper guard*, respectively (both, variables and coefficients, may differ for different guards). The *condition* $\varphi^{\le}$ is a conjunction of lower guards, and the condition $\varphi^{>}$ is a conjunction of upper guards. Rules may increase shared variables. We model this using an update vector $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$, which is added to the vector of shared variables, when the rule is executed. Then $\mathcal{R}$ is the finite set of rules.

Definition 1 *(Precedence).* For a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *precedence relation* $\prec_P$ as subset of $\mathcal{R} \times \mathcal{R}$ as follows:

$$r_1 \prec_P r_2 \text{ iff } r_1.to = r_2.from.$$

We denote by $\prec_P^+$ the transitive closure of $\prec_P$. If $r_1 \prec_P^+ r_2 \wedge r_2 \prec_P^+ r_1$, or if $r_1 = r_2$, we write $r_1 \sim_P r_2$.

The precedence relation thus captures the control flow of a process. For instance, in the example of Fig. 1 it captures that a process must execute rule r_4 before it can execute rule r_5.

Definition 2 *(Unlock relation).* For a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *unlock relation* $\prec_U$ as subset of $\mathcal{R} \times \mathcal{R}$ as follows: $r_1 \prec_U r_2$ iff there is a $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ and a $\mathbf{p} \in \mathbf{P}_{RC}$ satisfying

- $(\mathbf{g}, \mathbf{p}) \models r_1.\varphi^{\le} \wedge r_1.\varphi^{>}$,
- $(\mathbf{g}, \mathbf{p}) \not\models r_2.\varphi^{\le} \wedge r_2.\varphi^{>}$, and
- $(\mathbf{g} + r_1.\mathbf{u}, \mathbf{p}) \models r_2.\varphi^{\le} \wedge r_2.\varphi^{>}$.

In the example of Fig. 1, rule r_4 increases the shared variable x, and by that may unlock rule r_1. We thus write $r_4 \prec_U r_1$. Similarly, r_2 unlocks r_5 in the example.

Definition 3 *(Lock relation).* For a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *lock relation* $\prec_L$ as subset of $\mathcal{R} \times \mathcal{R}$ as follows: $r_1 \prec_L r_2$ iff there is a $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ and a $\mathbf{p} \in \mathbf{P}_{RC}$ satisfying

- $(\mathbf{g}, \mathbf{p}) \models r_1.\varphi^{\le} \wedge r_1.\varphi^{>}$,
- $(\mathbf{g}, \mathbf{p}) \models r_2.\varphi^{\le} \wedge r_2.\varphi^{>}$, and
- $(\mathbf{g} + r_1.\mathbf{u}, \mathbf{p}) \not\models r_2.\varphi^{\le} \wedge r_2.\varphi^{>}$.

Our analysis in Section 4 will show that only two types of conditions of the threshold automaton contribute to the diameter we are interested in. First, these are the conditions that appear in a rule r that can be unlocked by a rule r' that comes after rule r or is parallel to r in the control flow. More precisely, rule r' does not appear before r in the control flow. The other conditions we are interested in are those that appear in a rule r that can be locked by a rule r'' that is before r or parallel to r in the control flow; more precisely, r'' does not appear after r in the control flow. This leads to the definition of the following quantities.

Definition 4 *(Number of relevant conditions).* Given a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the following quantities:

$$\mathcal{C}^{\le} = |\{r.\varphi^{\le} : r \in \mathcal{R}, \exists r' \in \mathcal{R}.\, r' \not\prec_P^+ r \ \wedge\ r' \prec_U r\}|$$
$$\mathcal{C}^{>} = |\{r.\varphi^{>} : r \in \mathcal{R}, \exists r'' \in \mathcal{R}.\, r \not\prec_P^+ r'' \ \wedge\ r'' \prec_L r\}|$$
$$\mathcal{C} = \mathcal{C}^{\le} + \mathcal{C}^{>}.$$

To determine these quantities, we have to check whether a specific condition can potentially lock (or unlock) another one, as defined in Definition 2 (or Definition 3). Observe that this can be done efficiently using an SMT solver.

We consider specific threshold automata, namely those that naturally capture FTDAs, where rules that form cycles do not increase shared variables.

Definition 5 *(Canonical threshold automaton).* A threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ is canonical, if $r.\mathbf{u} = \mathbf{0}$ for all rules $r \in \mathcal{R}$ that satisfy $r \prec_P^+ r$.

The relation $\sim_P$ defines equivalence classes of rules. For a given set of rules $\mathcal{R}$ let $\mathcal{R}/\sim$ be the set of equivalence classes defined by $\sim_P$. We denote by $[r]$ the equivalence class of rule r. For two classes c_1 and c_2 from $\mathcal{R}/\sim$ we write $c_1 \prec_C c_2$ iff there are two rules r_1 and r_2 in $\mathcal{R}$ satisfying $[r_1] = c_1$ and $[r_2] = c_2$ and $r_1 \prec_P^+ r_2$ and $r_1 \not\sim_P r_2$. Observe that the relation $\prec_C$ is a strict partial order (irreflexive and transitive). Hence, there are linear extensions of $\prec_C$. Below, we fix an *arbitrary* of these linear extensions. We will later use it to sort transitions in a schedule.

Notation. We denote by $\prec_C^{lin}$ a linear extension of $\prec_C$.

Proposition 6. *If* $[r_2] \prec_C^{lin} [r_1]$ *then* $r_1 \not\prec_P^+ r_2$.

Proof. Suppose by contradiction that $[r_2] \prec_C^{lin} [r_1]$ and $r_1 \prec_P^+ r_2$. We derive a contradiction by distinguishing two cases:

- if $r_1 \not\sim_P r_2$, then by the definition of $\prec_C$, we get $[r_1] \prec_C [r_2]$;
- otherwise, that is, if $r_1 \sim_P r_2$, it follows that $[r_1] = [r_2]$.

In both cases we derive a contradiction to $[r_2] \prec_C^{lin} [r_1]$. □

The semantics of threshold automata are defined with respect to counter systems in the following section.

3.2. Counter systems

Given a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ and admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, we define in the following a counter system as a transition system (Σ, I, R) that consists of the set of configurations Σ, the set of initial configurations I, and the transition relation R.

Configurations A configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ consists of a vector of *counter values* $\sigma.\boldsymbol{\kappa} \in \mathbb{N}_0^{|\mathcal{L}|}$, a vector of *shared variable values* $\sigma.\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$, and a vector of *parameter values* $\sigma.\mathbf{p} = \mathbf{p}$. The set Σ is the set of all configurations. The function $N : \mathbf{P}_{RC} \to \mathbb{N}_0$ formalizes the number of processes to be modeled. In our example, the number of processes is given by the value of the parameter n. In [14], we discussed a case study where $N(n, t, f) = n - f$. The set of initial configurations I contains the configurations that satisfy

- $\sigma.\mathbf{g} = \mathbf{0}$,
- $\sum_{i \in \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = N(\mathbf{p})$, and
- $\sum_{i \notin \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = 0$.

Transition relation A *transition* is a pair $t = (rule, factor)$ of a rule of the threshold automaton and a non-negative integer called the *acceleration factor*, or just factor for short. To simplify notation, for a transition $t = (rule, factor)$ we refer by $t.\mathbf{u}$ and $t.\varphi^{>}$ etc. to $rule.\mathbf{u}$ and $rule.\varphi^{>}$ etc., respectively. We say a transition t is *unlocked* in configuration σ if $\forall k \in \{0, \ldots, t.factor - 1\}.\ (\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + k \cdot t.\mathbf{u}, \sigma.\mathbf{p}) \models t.\varphi^{\le} \wedge t.\varphi^{>}$. For transitions t_1 and t_2 we say that the two transitions are related iff $t_1.rule$ and $t_2.rule$ are related, e.g., $t_1 \prec_P t_2$ iff $t_1.rule \prec_P t_2.rule$.

A transition t is *applicable to (or enabled in)* configuration σ, if it is unlocked, and if $\sigma.\boldsymbol{\kappa}[t.from] \ge t.factor$. We say that σ' is the result of applying the (enabled) transition t to σ, and use the notation $\sigma' = t(\sigma)$, if

- t is enabled in σ
- $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.factor \cdot t.\mathbf{u}$
- $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
- if $t.from \neq t.to$ then
  - $\sigma'.\boldsymbol{\kappa}[t.from] = \sigma.\boldsymbol{\kappa}[t.from] - t.factor$,
  - $\sigma'.\boldsymbol{\kappa}[t.to] = \sigma.\boldsymbol{\kappa}[t.to] + t.factor$, and
  - for all ℓ in $\mathcal{L} \setminus \{t.from, t.to\}$ it holds that $\sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$
- if $t.from = t.to$ then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$

The transition relation $R \subseteq \Sigma \times \Sigma$ of the counter system is defined as follows: $(\sigma, \sigma') \in R$ iff there is a $r \in \mathcal{R}$ and a $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$. As updates to shared variables do not decrease their values, we obtain:

Proposition 7. *For all configurations σ, all rules r, and all transitions t applicable to σ, the following holds:*

1. *If $\sigma \models r.\varphi^{\leq}$ then $t(\sigma) \models r.\varphi^{\leq}$*
2. *If $t(\sigma) \not\models r.\varphi^{\leq}$ then $\sigma \not\models r.\varphi^{\leq}$*
3. *If $\sigma \not\models r.\varphi^{>}$ then $t(\sigma) \not\models r.\varphi^{>}$*
4. *If $t(\sigma) \models r.\varphi^{>}$ then $\sigma \models r.\varphi^{>}$*

The proposition formalizes a crucial property of our systems that will allow us to bound the diameter below: For instance, by repeated application of points 1 and 2 we obtain that once a condition of form $\varphi^{\leq}$ evaluates to true it will always do so in the future, while if $\varphi^{\leq}$ evaluates to false, it always has in the past. We conclude that for each condition, the evaluation changes at most once.

Schedules A *schedule* is a sequence of transitions. A schedule $\tau = t_1, \ldots, t_m$ is called *applicable* to configuration σ_0, if there is a sequence of configurations $\sigma_1, \ldots, \sigma_m$ such that $\sigma_i = t_i(\sigma_{i-1})$ for $0 < i \leq m$. A schedule $t_1, \ldots, t_m$ where $t_i.factor = 1$ for $0 < i \leq m$ is a *conventional schedule*. If there is a $t_i.factor > 1$, then a schedule is called *accelerated*.

We write $\tau \cdot \tau'$ to denote the concatenation of two schedules τ and τ', and treat a transition t as schedule. If $\tau = \tau_1 \cdot t \cdot \tau_2 \cdot t' \cdot \tau_3$, for some τ_1, τ_2, and τ_3, we say that transition t precedes transition t' in τ, and denote this by $t \to_\tau t'$.

4. Diameter of counter systems

In this section, we will present the outline of the proof of our main theorem:

Theorem 8. *Given a canonical threshold automaton* TA*, for each $\mathbf{p}$ in $\mathbf{P}_{RC}$ the diameter of the counter system is less than or equal to $d(\mathsf{TA}) = (\mathcal{C} + 1) \cdot |\mathcal{R}| + \mathcal{C}$, and thus independent of $\mathbf{p}$.*

From the theorem it follows that for all parameter values, reachability in the counter system can be verified by exploring runs of length at most $d(\mathsf{TA})$. However, the theorem alone is not sufficient to solve the parameterized model checking problem. For this, we combine the bound with the abstraction method in [14]. More precisely, the counter abstraction in [14] simulates the counter systems for *all* parameter values that satisfy the resilience condition. Consequently, the bound on the length of the run of the counter systems entails a bound for the counter abstraction. As we explain in Section 4.5, we exploit this in the experiments in Section 5.

4.1. Proof idea

Given a rule r, a schedule τ and two transitions t_i and t_j, with $t_i \to_\tau t_j$, the subschedule $t_i \cdot \ldots \cdot t_j$ of $\tau = t_1, \ldots, t_m$ is a *batch of rule* r if $t_\ell.rule = r$ for $i \leq \ell \leq j$, and if the subschedule is maximal, that is, $i = 1 \vee t_{i-1} \neq r$ and $j = m \vee t_{j+1} \neq r$. Similarly, we define a batch of a class c as a subschedule $t_i \cdot \ldots \cdot t_j$ where $[r_\ell] = c$ for $i \leq \ell \leq j$, and where the subschedule is maximal as before.

Definition 9 *(Sorted schedule).* Given a schedule τ, and the relation $\prec_C^{lin}$, we define $sort(\tau)$ as the schedule that satisfies:

1. $sort(\tau)$ is a permutation of schedule τ;
2. two transitions from the same equivalence class maintain their relative order, that is, if $t \to_\tau t'$ and $t \sim_P t'$, then $t \to_{sort(\tau)} t'$;
3. if $t \to_{sort(\tau)} t'$, then $t \sim_P t'$ or $[t] \prec_C^{lin} [t']$.

Proposition 10. *Given a schedule τ, and the relation $\prec_C^{lin}$, for each equivalence class defined by $\sim_P$ there is at most one batch in $sort(\tau)$.*

Proof. Assume by contradiction that there are two batches, that is, there is an equivalence class c such that there are two transitions t_1 and t_2 in c, and there is a transition $t' \notin c$ such that $t_1 \to_{sort(\tau)} t'$ and $t' \to_{sort(\tau)} t_2$. By Definition 9(3) it follows from $t_1 \to_{sort(\tau)} t'$ that $c \prec_c^{lin} [t']$ and from $t' \to_{sort(\tau)} t_2$ that $[t'] \prec_c^{lin} c$. As $\prec_c^{lin}$ is a total order we arrive at the required contradiction. □

Note that from Proposition 10 and Definition 9 (Points 1 and 2) it follows that $sort(\tau)$ is indeed unique for a given τ.

The crucial observation to prove Theorem 8 is that if we have a schedule $\tau_1 = t \cdot t'$ applicable to configuration σ with $t.rule = t'.rule$, we can replace it with another applicable (one-transition) schedule $\tau_2 = t''$, with $t''.rule = t.rule$ and $t''.factor = t.factor + t'.factor$, such that $\tau_1(\sigma) = \tau_2(\sigma)$. Thus, we can reach the same configuration with a shorter schedule. More generally, we may replace a batch of a rule by a single accelerated transition whose factor is the sum of all factors in the batch.

To bound the diameter, we have to bound the distance between any two configurations σ and σ' for which there is a schedule τ applicable to σ satisfying $\sigma' = \tau(\sigma)$. A simple case is if $sort(\tau)$ is applicable to σ and each equivalence class defined by the precedence relation consists of a single rule (e.g., the threshold automaton is a directed acyclic graph). Then by Proposition 10 we have at most $|\mathcal{R}|$ batches in $sort(\tau)$, that is, one per rule. By the reasoning of above we can replace each batch by a single accelerated transition.

However, in general $sort(\tau)$ may not be applicable to σ, or there are equivalence classes containing multiple rules, i.e., rules form cycles in the precedence relation. The first issue comes from locking and unlocking, and as discussed in Section 2, we identify milestone transitions, and show that we can apply *sort* to the segments between milestones in Section 4.3. We also deal with the issue of cycles in the precedence relation. It is ensured by *sort* that within a segment, all transitions that belong to a cycle form a batch. In Section 4.2, we replace such a batch by a batch where the remaining rules do not form a cycle. Removing cycles requires the assumption that shared variables are not incremented in cycles.

4.2. Dealing with cycles

We consider the distance between two configurations σ and σ' that satisfy $\sigma.\mathbf{g} = \sigma'.\mathbf{g}$, i.e., along any schedule connecting these configurations, the values of shared variables are unchanged, and so are thus the evaluations of guards. By Definition 5, we can apply this section's result to batches of a class of canonical threshold automata. The following definition captures how often processes go to and leave specific states, respectively, and the updates on the variables.

Definition 11. Given a schedule $\tau = t_1, t_2, \dots t_k$, we denote by $|\tau|$ the length k of the schedule. Further, we define the following vectors

$$\mathbf{in}(\tau)[\ell] = \sum_{\substack{1 \le i \le |\tau| \\ t_i.to = \ell}} t_i.factor,$$

$$\mathbf{out}(\tau)[\ell] = \sum_{\substack{1 \le i \le |\tau| \\ t_i.from = \ell}} t_i.factor,$$

$$\mathbf{up}(\tau) = \sum_{1 \le i \le |\tau|} t_i.\mathbf{u}.$$

From the definition of a counter system, we directly obtain:

Proposition 12. *For all configurations σ, and all schedules τ applicable to σ, if $\sigma' = \tau(\sigma)$, then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa} + \mathbf{in}(\tau) - \mathbf{out}(\tau)$, and $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + \mathbf{up}(\tau)$.*

The proposition directly implies the following:

Proposition 13. *For all configurations σ, and all schedules τ and τ' applicable to σ, if $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathbf{up}(\tau) = \mathbf{up}(\tau')$, then $\tau(\sigma) = \tau'(\sigma)$.*

Given a schedule $\tau = t_1, t_2, \dots$ we say that the index sequence $i(1), \dots, i(j)$ is a cycle in τ, if for all b, $1 \le b < j$, it holds that $t_{i(b)}.to = t_{i(b+1)}.from$, and $t_{i(j)}.to = t_{i(1)}.from$, and $t_{i(c)} \ne t_{i(d)}$ for $1 \le c < d \le j$. Let $\mathcal{R}(\tau)$ be the set of rules appearing in τ, that is, $\{r : t_i \in \tau \wedge t_i.rule = r\}$.

In the following proposition we are concerned with removing cycles, without considering applicability of the resulting schedule. We consider applicability later in Theorem 16.

Proposition 14. *For all schedules τ, if τ contains a cycle, then there is a schedule τ' satisfying $|\tau'| < |\tau|$, $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$.*

Proof. Let $I = i(1), \ldots, i(j)$ be a cycle in $\tau = t_1, t_2, \ldots$, let $\tau_I = t_{i(1)}, \ldots, t_{i(j)}$, and let θ be the schedule $t'_1, \ldots, t'_j$ satisfying for $1 \le k \le j$ that

- $t'_k.rule = t_{i(k)}.rule$, and
- $t'_k.factor = t_{i(k)}.factor - \min\{t_b.factor : b \in I\}$.

As I is a cycle, by definition, for each local state s, the number of transitions that go out of s is equal to the number of transitions that enter s, that is, $|\{b : b \in I, t_b.from = s\}| = |\{b : b \in I, t_b.to = s\}|$. As θ is constructed from τ_I by reducing the factor of each transition by $\min\{t_b.factor : b \in I\}$, it follows that:

$$\mathbf{in}(\theta) - \mathbf{out}(\theta) = \mathbf{in}(\tau_I) - \mathbf{out}(\tau_I) \tag{1}$$

Denote with $\tau[\theta/I]$ the schedule obtained from τ by substituting all transitions in the positions $i(1), i(2), \ldots, i(j)$ with the transitions $t'_1, t'_2, \ldots, t'_j$ of the schedule θ, respectively. From (1), we immediately conclude that $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau[\theta/I]) - \mathbf{out}(\tau[\theta/I])$. Further, by construction, the schedule θ contains at least one transition t'_k with $t'_k.factor = 0$ for $1 \le k \le j$. Let τ' be the schedule obtained from the schedule $\tau[\theta/I]$ by removing all the transitions in the positions $i(1), i(2), \ldots, i(j)$ that have *factor* equal to zero. As removal of such transitions does not change **in** and **out**, we immediately conclude that $\mathbf{in}(\tau') - \mathbf{out}(\tau') = \mathbf{in}(\tau[\theta/I]) - \mathbf{out}(\tau[\theta/I])$ and thus $\mathbf{in}(\tau') - \mathbf{out}(\tau') = \mathbf{in}(\tau) - \mathbf{out}(\tau)$. Moreover, as we have removed at least one transition, it holds that $|\tau'| < |\tau|$, and as we have not added new rules, it holds that $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$. □

Repeated application of the proposition leads to a cycle-free schedule (possibly the empty schedule), and we obtain:

Theorem 15. *For all schedules τ, there is a schedule τ' that contains no cycles, $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$.*

The issue with this theorem is that τ' is not necessarily applicable to the same configurations as τ. In the following theorem, we prove that if a schedule satisfies a specific condition on the order of transitions, then it is applicable.

Theorem 16. *Let σ and σ' be two configurations with $\sigma.\mathbf{g} = \sigma'.\mathbf{g}$, and τ be a schedule with $\mathbf{up}(\tau) = \mathbf{0}$, every transition t is unlocked in σ and satisfies $t.from \ne t.to$, and where if $t_i \to_\tau t_j$, then $t_j \not\prec_P t_i$. If $\sigma'.\boldsymbol{\kappa} - \sigma.\boldsymbol{\kappa} = \mathbf{in}(\tau) - \mathbf{out}(\tau)$, then τ is applicable to σ.*

Proof. The proof is by induction on $|\tau|$.
Base: $|\tau| = 0$. Follows trivially.
Step: $|\tau| > 0$. Let $\tau = t \cdot \tau'$ for some τ'. We first prove that t is applicable to σ, and then that $t(\sigma)$ and τ' satisfy the induction hypothesis. Then, the theorem follows.

By assumption, τ' does not contain a transition t' satisfying $t' \prec_P t$, that is, for all t_i in τ', $t_i.to \ne t.from$, and by assumption, $t.from \ne t.to$, and therefore

$$\mathbf{in}(\tau)[t.from] = 0 \tag{2}$$

Recall that from the definition of a configuration,

$$\sigma'.\boldsymbol{\kappa}[t.from] \ge 0. \tag{3}$$

By assumption

$$\sigma'.\boldsymbol{\kappa}[t.from] - \sigma.\boldsymbol{\kappa}[t.from] = \mathbf{in}(\tau)[t.from] - \mathbf{out}(\tau)[t.from].$$

Applying (3) we obtain $\sigma.\boldsymbol{\kappa}[t.from] \ge \mathbf{out}(\tau)[t.from] - \mathbf{in}(\tau)[t.from]$, and further from (2) we get $\sigma.\boldsymbol{\kappa}[t.from] \ge \mathbf{out}(\tau)[t.from]$. As t is in τ, from Definition 11 follows that $\mathbf{out}(\tau)[t.from] \ge t.factor$, and finally $\sigma.\boldsymbol{\kappa}[t.from] \ge t.factor$. It follows that t is applicable to σ.

It remains to prove that

$$\sigma'.\boldsymbol{\kappa} - t(\sigma).\boldsymbol{\kappa} = \mathbf{in}(\tau') - \mathbf{out}(\tau'), \tag{4}$$

which allows us to invoke the induction hypothesis. To do so, we consider the components of $\sigma.\kappa$. Observe that for all local states s, $s \notin \{t.from, t.to\}$, we have $t(\sigma).\boldsymbol{\kappa}[s] = \sigma.\boldsymbol{\kappa}[s]$, $\mathbf{in}(\tau')[s] = \mathbf{in}(\tau)[s]$, and $\mathbf{out}(\tau')[s] = \mathbf{out}(\tau)[s]$.

Hence, $\sigma'.\boldsymbol{\kappa}[s] - t(\sigma).\boldsymbol{\kappa}[s] = \mathbf{in}(\tau')[s] - \mathbf{out}(\tau')[s]$. To prove (4), it remains to consider the indices $t.from$ and $t.to$. Recall that by assumption, $t.from \ne t.to$.

Component $t.from$ of (4). The counter for $t.from$ changes, that is, $\sigma.\boldsymbol{\kappa}[t.from] = t(\sigma).\boldsymbol{\kappa}[t.from] + t.factor$. As τ' is obtained by removing t from τ, we have $\mathbf{out}(\tau)[t.from] = \mathbf{out}(\tau')[t.from] + t.factor$, and $\mathbf{in}(\tau)[t.from] = \mathbf{in}(\tau')[t.from]$.
From the assumption $\sigma'.\boldsymbol{\kappa} - \sigma.\boldsymbol{\kappa} = \mathbf{in}(\tau) - \mathbf{out}(\tau)$ it follows that

$$\sigma'.\boldsymbol{\kappa}[t.from] - t(\sigma).\boldsymbol{\kappa}[t.from] - t.factor = \mathbf{in}(\tau')[t.from] - \mathbf{out}(\tau')[t.from] - t.factor,$$

and the case follows.

Component $t.to$ of (4). The counter for $t.to$ changes, that is, $\sigma.\boldsymbol{\kappa}[t.to] = t(\sigma).\boldsymbol{\kappa}[t.to] - t.factor$. As τ' is obtained by removing t from τ, we have $\mathbf{in}(\tau)[t.to] = \mathbf{in}(\tau')[t.from] + t.factor$ and $\mathbf{out}(\tau)[t.to] = \mathbf{out}(\tau')[t.to]$.
Again, it follows from the assumption that

$$\sigma'.\boldsymbol{\kappa}[t.to] - t(\sigma).\boldsymbol{\kappa}[t.to] + t.factor = \mathbf{in}(\tau')[t.to] + t.factor - \mathbf{out}(\tau')[t.to].$$

Hence $\sigma'.\boldsymbol{\kappa} - t(\sigma).\boldsymbol{\kappa} = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and the theorem follows. □

Given a configuration σ, and a schedule τ applicable to σ, with $\mathbf{up}(\tau) = \mathbf{0}$, by Theorem 15 there is a cycle-free schedule τ' with $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$. As τ' contains no cycle, we may re-order the transitions in τ' according to $\prec_P$, as required by Theorem 16, such that there is at most one block per rule. By Theorem 16, the resulting schedule is applicable, and we obtain:

Corollary 17. *For all configurations σ, and all schedules τ applicable to σ, with $\mathbf{up}(\tau) = \mathbf{0}$, there is a schedule with at most one batch per rule applicable to σ satisfying that τ' contains no cycles, $\tau'(\sigma) = \tau(\sigma)$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$.*

4.3. Defining milestones and swapping transitions

In this section we deal with locking and unlocking. To this end, we define milestones, and show that transitions that are not milestones can be swapped.

Proposition 18. *For all configurations σ, and all transitions t_1 and t_2, if t_2 is applicable to $t_1(\sigma)$ and t_1 is applicable to $t_2(\sigma)$, then $t_2(t_1(\sigma)) = t_1(t_2(\sigma))$.*

Proof. Follows from commutativity of addition applied to counters and shared variables. □

As discussed in Section 4.1, we would like to replace a schedule (or subschedule) τ by $sort(\tau)$, so that the resulting schedule $sort(\tau)$ is applicable. To do so, we have to show that if we start with τ and swap adjacent transitions until we reach $sort(\tau)$, all the intermediate schedules and the final schedule are applicable. However, due to locking and unlocking, we cannot always swap transitions. For instance, if t' appears directly before t in a schedule, and t' unlocks t (that is, t is locked in the configuration in which t' is applied and unlocked after the application of t'), swapping t' and t leads to a schedule which is not applicable. This is because t is not applicable. We observe that this problem occurs

- because we want to swap t' and t (t is before t' in the linear extension of the precedence relation, that is, t' is not before t in the precedence relation),
- t' unlocks t, and
- t is locked in the beginning.

In such cases, t must not be moved "to the left" in the schedule, and we call t a *left milestone*. We capture this intuition in the following definition.

Definition 19 *(Left milestone).* Given a configuration σ and a schedule $\tau = \tau' \cdot t \cdot \tau''$ applicable to σ, the transition t is a left milestone for σ and τ, if

1. there is a transition t' in τ' satisfying $t' \not\prec_P^+ t \wedge t' \prec_U t$,
2. $t.\varphi^{\le}$ is locked in σ, and
3. for all t' in τ' it holds that $t'.\varphi^{\le} \neq t.\varphi^{\le}$.

The following definition of right milestones is analogous but, instead of unlocking and considering transitions that are locked in the beginning, considers the locking relation and transitions that are locked after application of the schedule.

Definition 20 *(Right milestone).* Given a configuration σ and a schedule $\tau = \tau' \cdot t \cdot \tau''$ applicable to σ, the transition t is a right milestone for σ and τ, if

1. there is a transition t'' in τ'' satisfying $t \not\prec_P^+ t'' \wedge t'' \prec_L t$,
2. $t.\varphi^>$ is locked in $\tau(\sigma)$, and
3. for all t'' in τ'' it holds that $t''.\varphi^> \neq t.\varphi^>$.

Milestones divide schedules into segments that are defined as follows.

Definition 21 *(Segment).* Given a schedule τ and configuration σ, τ' is a segment if it is a subschedule of τ, and does not contain a milestone for σ and τ.

The following theorem shows that two transitions that are not milestones can be swapped. Together with the fact that by definition the number of milestones is bounded by $\mathcal{C}$, repeated application of the theorem eventually leads to a schedule where milestones and sorted schedules alternate.

Theorem 22. *Let σ be a configuration, τ a schedule applicable to σ, and $\tau = \tau_1 \cdot t_1 \cdot t_2 \cdot \tau_2$. If transitions t_1 and t_2 are not milestones for σ and τ, and satisfy $[t_2] \prec_C^{lin} [t_1]$, then*

i. schedule $\tau' = \tau_1 \cdot t_2 \cdot t_1 \cdot \tau_2$ is applicable to σ, and
ii. $\tau'(\sigma) = \tau(\sigma)$.

Proof. We prove (i) by showing that (a) t_2 is applicable to $\sigma' = \tau_1(\sigma)$, and (b) t_1 is applicable to $t_2(\sigma')$. From (a), (b), and Proposition 18, Point (i) then follows.
(a) We prove that t_2 is applicable to σ' by case distinction

- $t_1 \not\prec_U t_2$. As the rule of t_1 never unlocks the rule of t_2, and because t_2 is unlocked in $t_1(\sigma')$, t_2 must also be unlocked in σ' due to Proposition 7.
- Otherwise, that is, $t_1 \prec_U t_2$. Due to Proposition 6, from $[t_2] \prec_C^{lin} [t_1]$ follows that $t_1 \not\prec_P^+ t_2$. It follows that t_2 satisfies Definition 19(1).
  Now assume by way of contradiction that t_2 is locked in σ'. We will show that from this assumption it follows that t_2 is a left milestone[2] for σ and τ to derive a contradiction and conclude that t_2 is unlocked in σ': As t_2 is locked in σ', from repeated application of Proposition 7(2) we obtain that $t_2.\varphi^\leq$ is locked in σ, and all intermediate configurations until σ'. As it is locked in σ, transition t_2 satisfies Definition 19(2). As the transition is locked in all intermediate configurations no transition that is guarded with the same condition can appear in the prefix, that is, for each transition t' in τ_1 it holds that $t'.\varphi^\leq \neq t_2.\varphi^\leq$, which satisfies Definition 19(3). Hence, t_2 is a left milestone, which contradicts that t_2 is not a milestone. We conclude that t_2 is unlocked in σ'.

As t_2 is unlocked in σ', by the definition of applicability, it is sufficient to prove that $\sigma'.\boldsymbol{\kappa}[t_2.from] \geq t_2.factor$. By assumption, t_2 is applicable to $t_1(\sigma')$ so that by the definition of applicability

$$t_1(\sigma').\boldsymbol{\kappa}[t_2.from] \geq t_2.factor \tag{5}$$

We have to distinguish two cases:

- If $t_1.from = t_2.from$, then $t_1(\sigma').\boldsymbol{\kappa}[t_2.from] = \sigma'.\boldsymbol{\kappa}[t_2.from] - t_1.factor$. From (5) it follows that $\sigma'.\boldsymbol{\kappa}[t_2.from] \geq t_1.factor + t_2.factor$ and this case follows.
- Otherwise, that is, if $t_1.from \neq t_2.from$. Due to Proposition 6, from $[t_2] \prec_C^{lin} [t_1]$ follows that $t_1 \not\prec_P^+ t_2$. From $t_1 \not\prec_P^+ t_2$ follows that $t_1 \not\prec_P t_2$ and thus $t_1.to \neq t_2.from$. Hence, $t_1(\sigma').\boldsymbol{\kappa}[t_2.from] = \sigma'.\boldsymbol{\kappa}[t_2.from]$. Consequently, this case follows at once from (5).

(b) As t_1 is applicable to σ', it is unlocked in σ'. We again distinguish two cases:

- $t_2 \not\prec_L t_1$. As the rule of t_2 never locks the rule of t_1, and because t_1 is unlocked in σ', the transition t_1 must also be unlocked in $t_2(\sigma')$.
- Otherwise, that is, $t_2 \prec_L t_1$. Due to Proposition 6, from $[t_2] \prec_C^{lin} [t_1]$ follows that $t_1 \not\prec_P^+ t_2$. Hence, t_1 satisfies Definition 20(1). Now assume by way of contradiction that t_1 is locked in $t_2(\sigma')$. We will show that t_1 is a right milestone to arrive at the required contradiction: As t_1 is unlocked in σ and locked in $t_2(\sigma')$, it is locked due to $t_1.\varphi^>$, which evaluates to false in $t_2(\sigma')$. As t_1 is locked in $t_2(\sigma')$, and as the values of global variables in $t_2(t_1(\sigma'))$ are greater than or equal to those of $t_2(\sigma')$, it follows that $t_1.\varphi^>$ evaluates to false in $t_2(t_1(\sigma'))$. From this and repeated application

[2] Intuitively, as τ' is obtained from τ by moving t_2 one position to the left, we argue about t_2 being a left milestone here. In point (b) below, we view τ' being obtained from τ by moving t_1 one position to the right, and consequently derive a contradiction using the notion of right milestone for t_1.

of Proposition 7(3) we obtain that $t_1.\varphi^{>}$ is locked in $\tau(\sigma)$, which satisfies Definition 20(2). Further as t_1 is locked in $t_2(\sigma')$, for each transition t'' in τ_2 it holds that $t''.\varphi^{>} \neq t_2.\varphi^{>}$, which satisfies Definition 20(3). Hence, t_1 is a right milestone, which provides the required contradiction.

We conclude that t_1 is unlocked in $t_2(\sigma')$.

It remains to show that $t_2(\sigma').\boldsymbol{\kappa}[t_1.from] \geq t_1.factor$, which can be proven analogously to the argument on counters in (a).

By assumption, τ_2 is applicable to $t_2(t_1(\tau_1(\sigma)))$, and from Proposition 18 follows that $t_2(t_1(\tau_1(\sigma))) = t_1(t_2(\tau_1(\sigma)))$. Consequently, τ_2 is applicable to $t_1(t_2(\tau_1(\sigma)))$. Hence, τ' is applicable to σ, and Point (ii) of the theorem statement follows from Proposition 13. □

4.4. Proof of main theorem

Theorem 8. *Given a canonical threshold automaton* TA*, for each* $\mathbf{p}$ *in* $\mathbf{P}_{RC}$ *the diameter of the counter system is less than or equal to* $d(\mathsf{TA}) = (\mathcal{C}+1)\cdot|\mathcal{R}|+\mathcal{C}$*, and thus independent of* $\mathbf{p}$*.*

Proof. We can view a schedule τ applicable to σ as alternation of segments τ_i and milestones m_i. We obtain from repeated application of Theorem 22, that each schedule applicable to σ can be transformed into a schedule $sort(\tau_1)\cdot m_1 \cdot sort(\tau_2)\cdot m_2 \cdot \ldots$ that is also applicable to σ. By Proposition 10 there is at most one batch per equivalence class in $sort(\tau_i)$. If this equivalence class consists of a single rule, the batch can be replaced by a single (accelerated) transition. Otherwise, that is, if a class consists of say x rules, as we consider canonical threshold automata that do not have updates to shared variables in rules r with $r \prec_P^+ r$, we can use the construction of Section 4.2 to replace the batch of this class by at most x accelerated transitions. We arrive at a segment that contains at most one transition per rule, that is, at most $|\mathcal{R}|$ transitions. It remains to bound the number of milestones.

As by Definition 19(3) and Definition 20(3) there is at most one milestone per condition, we have at most $\mathcal{C}$ milestones as defined in Definition 4. To conclude, we obtain an accelerated schedule, consisting of $\mathcal{C}$ milestones and $\mathcal{C}+1$ segments of length at most $|\mathcal{R}|$. □

4.5. Applying our result

In the proof of Theorem 8, we bound the length of all segments by $|\mathcal{R}|$. However, by Definition 19, segments to the left of a left milestone cannot contain transitions for rules with the same condition as the milestone. The same is true for segments to the right of right milestones. As we will see in Section 5.4, our tool ByMC explores all orders of milestones, an uses this observation about milestones to compute a more precise bound $\boldsymbol{d}^{\star}$ for the diameter.

Our encoding of the counter abstraction only increments and decrements counters. If $|\hat{D}|$ is the size of the abstract domain, a transition in a counter system is simulated by at most $|\hat{D}|-1$ steps in the counter abstraction; this leads to the diameter $\hat{\boldsymbol{d}}$ for counter abstractions, which we use in our experiments.

5. Experimental evaluation

We have implemented the techniques discussed in this article in our tool ByMC [16]. The input are the descriptions of our benchmarks given in our parametric extension of Promela [15], which describe parameterized processes. Hence, as preliminary step, ByMC computes the PIA data abstraction [14] in order to obtain finite state processes. Based on this, ByMC does preprocessing to compute threshold automata and the locking and unlocking relations, and to generate the inputs for our model checking back-ends.

5.1. Preprocessing

To apply our results, we have to compute the set of rules $\mathcal{R}$. Recall that a rule is a tuple $(from, to, \varphi^{\leq}, \varphi^{>}, \mathbf{u})$. ByMC computes the reachable local states. In the case of the CBC case study, e.g., this step reduces the local states under consideration from 2000 to 100, approximately. All our experiments — including the ones with FASTer [2] — are based on the reduced local state space.

Then, for each pair $(from, to)$, ByMC explores symbolic paths to compute the guards and update vectors for the pair, and removes the infeasible paths using an SMT solver. From this we get the set of rules $\mathcal{R}$. Then, ByMC encodes Definition 2 in the SMT solver Yices, to construct the lock $\prec_L$ and unlock $\prec_U$ relations. ByMC computes the relations $\{(r,r') : r' \not\prec_P^+ r \wedge r' \prec_U r\}$ and $\{(r,r'') : r \not\prec_P^+ r'' \wedge r'' \prec_L r\}$ as required by Definition 4. This provides the bounds we use for bounded model checking.

Table 1
Benchmark overview giving the article from which we formalized the distributed algorithm, the number of shared variables, and the model and the size of the abstract domain.

Benchmark	Reference	Shared vars	Abs. domain size
FRB	[9]	1	2
STRB	[32]	1	4
ABA0	[8]	2	4
ABA1	[8]	2	5
CBC0	[27]	4	4
CBC1	[27]	4	5
NBAC(C)	[31]	4	4

5.2. Back-ends

ByMC generates the PIA counter abstraction [14] to be used by the following back-end model checkers. We have also implemented an automatic abstraction refinement loop for the counterexamples provided by NuSMV.

BMC. NuSMV 2.5.4 [10] (using MiniSAT) performs incremental bounded model checking with the bound $\hat{d}$. If a counterexample is reported, ByMC refines the system as explained in [14], if the counterexample is spurious.

BMCL. This technique combines the power of NuSMV 2.5.4 and of the state-of-the-art multi-core SAT solver Plingeling ats1 [4]. NuSMV performs incremental bounded model checking for 30 steps. If a spurious counterexample is found, then ByMC refines the system description. When NuSMV does not report a counterexample, NuSMV generates a single CNF with the bound $\hat{d}$. Satisfiability of this formula is then checked with Plingeling.

BDD. NuSMV 2.5.4 performs BDD-based symbolic checking.

FAST. FASTer 2.1 [2] performs reachability analysis using the plugin Mona-1.3.

5.3. Benchmarks

We encoded several asynchronous FTDAs in our parametric Promela, following the technique in [15]; they can be obtained from our git repository.[3] All models contain transitions with lower threshold guards. The benchmarks CBC also contain upper threshold guards. If we ignore self-loops, the precedence relation of all but NBAC and NBACC, which have non-trivial cycles, are partial orders. We provide the most relevant data on these benchmarks in Table 1, and discuss them in more detail below.

Folklore reliable broadcast (FRB) In this FTDA, n processes have to agree on whether a process has broadcast a message, in the presence of $f \le n$ crashes. Our model of FRB has one shared variable and the abstract domain of two intervals $[0, 1)$ and $[1, \infty)$. In this paper, we are concerned with the safety property *unforgeability*: If no process is initialized with value 1 (message from the broadcaster), then no correct process ever accepts.

Consistent broadcast (STRB) Here, we have $n - f$ correct processes and $f \ge 0$ Byzantine faulty ones. The resilience condition is $n > 3t \wedge t \ge f$. There is one shared variable and the abstract domain of four intervals $[0, 1)$, $[1, t + 1)$, $[t + 1, n - t)$, and $[n - t, \infty)$. In the experiments reported here, we check only unforgeability (see FRB), whereas in [14] we checked also liveness properties.

Byzantine agreement (ABA) There are $n > 3t$ processes, $f \le t$ of them Byzantine faulty. The model has two shared variables. We have to consider two different cases for the abstract domain, namely, case ABA0 with the domain $[0, 1)$, $[1, t + 1)$, $[t+1, \lceil\frac{n+t}{2}\rceil)$, and $[\lceil\frac{n+t}{2}\rceil, \infty)$ and case ABA1 with the domain $[0, 1)$, $[1, t+1)$, $[t+1, 2t+1)$, $[2t+1, \lceil\frac{n+t}{2}\rceil)$, and $[\lceil\frac{n+t}{2}\rceil, \infty)$. As for FRB, we check unforgeability. This case study, and all below, run out of memory when using Spin for model checking the counter abstraction [14].

Condition-based consensus (CBC) This is a restricted variant of consensus solvable in asynchronous systems. We consider the binary version of condition-based consensus in the presence of clean crashes, which requires four shared variables. Under the resilience condition $n > 2t \wedge f \ge 0$, we have to consider two different cases depending on f: If $f = 0$ we have case CBC0 with the domain $[0, 1)$, $[1, \lceil\frac{n}{2}\rceil)$, $[\lceil\frac{n}{2}\rceil, n - t)$, and $[n - t, \infty)$. If $f \ne 0$, case CBC1 has the domain: $[0, 1)$, $[1, f)$, $[f, \lceil\frac{n}{2}\rceil)$, $[\lceil\frac{n}{2}\rceil, n - t)$, and $[n - t, \infty)$. We verified several properties, all of which resulted in experiments with similar characteristics. We only give *validity*$_0$ in the table, i.e., no process accepts value 0, if all processes initially have value 1.

[3] https://github.com/konnov/fault-tolerant-benchmarks/tree/master/concur14.

Table 2
Summary of experiments on AMD Opteron®Processor 6272 with 192 GB RAM and 32 CPU cores. Plingeling used up to 16 cores. "TO" denotes timeout of 24 hours; "OOM" denotes memory overrun of 64 GB; "ERR" denotes runtime error; "RTO" denotes that the refinement loop timed out. When BMC and BMCL times out, we indicate the maximum length of the explored computations in percentage of the predicted diameter bound.

Input FTDA	Threshold A.				Bounds			Time, [HH:]MM:SS				Memory, GB			
	$\vert\mathcal{L}\vert$	$\vert\mathcal{R}\vert$	$\mathcal{C}^{\le}$	$\mathcal{C}^{>}$	d	$d^{\star}$	$\hat{d}$	BMCL	BMC	BDD	FAST	BMCL	BMC	BDD	FAST
Fig. 1	5	5	1	0	11	9	27	00:00:03	00:00:04	00:01	00:00:08	0.01	0.02	0.02	0.06
FRB	6	8	1	0	17	10	10	00:00:13	00:00:13	00:06	00:00:08	0.01	0.02	0.02	0.01
STRB	7	15	3	0	63	30	90	00:00:09	00:00:06	00:04	00:00:07	0.02	0.03	0.02	0.07
ABA0	37	180	6	0	1266	586	1758	00:21:26	02:20:10	00:15	00:08:40	6.37	1.49	0.07	3.56
ABA1	61	392	8	0	3536	1655	6620	TO 25%	TO 12%	00:33	02:36:25	TO	TO	0.08	15.65
CBC0	43	204	0	0	204	204	612	01:38:54	TO 57%	OOM	ERR	1.28	TO	OOM	ERR
CBC1	115	896	1	1	2690	2180	8720	TO 05%	TO 11%	TO	TO	TO	TO	TO	TO
NBACC	109	1724	6	0	12 074	5500	16 500	RTO	RTO	TO	TO	RTO	RTO	TO	TO
NBAC	77	1356	6	0	9498	4340	13 020	RTO	RTO	TO	TO	RTO	RTO	TO	TO
When a Bug is Introduced															
ABA0	32	139	6	0	979	469	1407	00:00:16	00:00:18	TO	00:05:57	0.04	0.04	TO	2.70
ABA1	54	299	8	0	2699	1305	5220	00:00:22	00:00:21	TO	ERR	0.06	0.06	TO	ERR

Non-blocking atomic commitment (NBAC and NBACC) Here, n processes are initialized with Yes or No and decide on whether to commit a transaction. The transaction must be aborted if at least one process is initialized to No. We consider the cases NBACC and NBAC of clean crashes and crashes, respectively. Both models contain four shared variables, and the abstract domain is $[0, 1)$ and $[1, n)$ and $[n - 1, n)$, and $[n, \infty)$. The algorithm uses a failure detector, which is modeled as local variable that changes its value non-deterministically.

5.4. Evaluation

Table 2 summarizes the experiments. For the threshold automata, we give the number of local states $|\mathcal{L}|$, the number of rules $|\mathcal{R}|$, and conditions according to Definition 4, i.e., $\mathcal{C}^{\le}$ and $\mathcal{C}^{>}$. The column $\boldsymbol{d}$ provides the bound on the diameter as in Theorem 8, whereas the column $\boldsymbol{d}^{\star}$ provides an improved diameter, and $\hat{d}$ the diameter of the counter abstraction, both discussed in Section 4.5.

As the experiments show, all techniques rapidly verify FRB, STRB, and Fig. 1. We had already verified FRB and STRB before using Spin [14]. The more challenging examples are ABA0 and ABA1, where BDD clearly outperforms the other techniques. Bounded model checking is slower here, because the diameter bound does not exploit knowledge on the specification. FAST performs well on these benchmarks. We believe this is because many rules are always disabled, due to the initial states as given in the specification. To confirm this intuition, we introduced a bug into ABA0 and ABA1, which allows the processes to non-deterministically change their value to 1. This led to a dramatic slowdown of BDD and FAST, as reflected in the last two lines.

Using the bounds of this paper, BMCL verified CBC0, whereas all other techniques failed. BMCL did not reach the bounds for CBC1 with our experimental setup. In this case, we report the percentages of the bounds we reached with bounded model checking.

In the experiments with NBAC and NBACC, the refinement loop timed out. We are convinced that we can address this issue by integrating the refinement loop with an incremental bounded model checker.

While we could not check all the benchmarks with the technique of this paper, a more aggressive offline partial order reduction in combination with SMT-based bounded model checking [17] allowed us to verify also these benchmarks.

6. Related work and discussions

Specific forms of counter systems can be used to model parameterized systems of concurrent processes. Lubachevsky [25] discusses *compact* programs that reach each state in a bounded number of steps, where the bound is independent of the number of processes. Besides, in [25] he gives examples of compact programs, and in [24] he proves that specific semaphore programs are compact. We not only show compactness, but give a bound on the diameter. In our case, communication is not restricted to semaphores, but we have threshold guards. Counter abstraction [30] follows this line of research, but as discussed by Basler et al. [3], does not scale well for large numbers of local states.

Another line of research is on *acceleration* in infinite state systems, e.g., in flat counter automata [22]. Acceleration is a technique that computes the transitive closure of a transition relation and applies it to the set of states. The tool FAST [1] uses the transitive closure of transitions to compute the set of reachable states in a symbolic procedure. This appears closely related to our transitions with acceleration factor. However, in [1] a transition is chosen and accelerated dynamically in the course of symbolic state space exploration, while we use acceleration factors and reordering to construct a bound as a formula over the characteristics of a threshold automaton (precedence, lock, and unlock relations). Our tool generates the cardinalities of these relations to compute length of computations for bounded model checking.

One can achieve completeness in bounded model checking by exploring all runs that are not longer than the diameter of the system [5]. This was later generalized to the notion of *completeness threshold* by Clarke et al. [11] in the presence of safety and liveness properties. To find a completeness threshold for a liveness property, it is sufficient to compute the diameter of the synchronous product of the transition system and a Büchi automaton, which represents the computations violating the property. As in general, computing the diameter is believed to be as hard as model checking, one can use a coarser bound provided by the reoccurrence diameter [19]. In practice, the reoccurrence diameter of counter abstraction is prohibitively large, so that we are interested bounds on the diameter.

Partial orders are a useful concept when reasoning about distributed systems [20]. In the context of model checking, *partial order reduction* [13,33,29] is a widely used technique to reduce the search space. It is based on the idea that changing the order of steps of concurrent processes leads to "equivalent" behavior with respect to the specification. Typically, partial order reduction is used on-the-fly to prune runs that are equivalent to representative ones. In contrast, in this paper, we bound the length of representative runs offline in order to ensure completeness of bounded model checking. Based on the ideas presented here, in [17] we introduce a more aggressive form of partial order reduction that, together with an encoding of a counter system in SMT, allowed us to verify reachability of even more involved fault-tolerant distributed algorithms. In the context of FTDAs, a partial order reduction was introduced by Bokor et al. [7]. Similar to this paper, they focus on "quorum transitions" that count messages. The technique by Bokor et al. [7] can be used for model checking small instances, while we focus on parameterized model checking.

Our technique of determining which transitions can be swapped in a run reminds of *movers* as discussed by Lipton [23], or more generally the idea to show that certain actions can be grouped into larger atomic blocks to simplify proofs [12,21]. However, movers address the issue of grouping many local transitions of a process together. In contrast, we conceptually group transitions of different processes together into one accelerated transition. Moreover, the definition of a mover by Lipton is independent of a specific run: a left mover (e.g., a "release" operation) is a transition that in *all runs* can "move to the left" with respect to transitions of other processes. In our work, we look at specific runs and identify which transitions (milestones) must not move in this run.

Our technique targets at threshold-based fault-tolerant distributed algorithms, that is, asynchronous distributed algorithms that communicate by sending messages to all and compare the number of received messages to linear combinations of parameters. As motivated by this application domain (and as discussed in the introduction), the systems we consider are symmetric, and the threshold automata we consider are restricted in that shared variables cannot be decreased, and rules that form a cycle in a threshold automaton may not increase shared variables. To model concurrent systems other than fault-tolerant distributed algorithms, it may be interesting to weaken the latter two restrictions. Our results on the diameter do not necessarily carry over to less restricted threshold automata and counter systems.

As next steps we will focus on liveness of fault-tolerant distributed algorithms. In fact the liveness specifications are in the fragment of linear temporal logic for which it is proven [18] that a formula can be translated into a cliquey Büchi automaton. For such automata, Kroening et al. provide a completeness threshold. Still, there are open questions related to applying our results to the idea by Kroening et al. [18].

References

[1] S. Bardin, A. Finkel, J. Leroux, L. Petrucci, FAST: acceleration from theory to practice, Int. J. Softw. Tools Technol. Transf. 10 (5) (2008) 401–424.
[2] S. Bardin, J. Leroux, G. Point, Fast extended release, in: CAV, in: LNCS, vol. 4144, 2006, pp. 63–66.
[3] G. Basler, M. Mazzucchi, T. Wahl, D. Kroening, Symbolic counter abstraction for concurrent software, in: CAV, in: LNCS, vol. 5643, 2009, pp. 64–78.
[4] A. Biere, Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013, Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions, 2013, pp. 51–52.
[5] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: TACAS, in: LNCS, vol. 1579, 1999, pp. 193–207.
[6] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, J. Widder, Decidability of parameterized verification, in: Synthesis Lectures on Distributed Computing Theory, 2015.
[7] P. Bokor, J. Kinder, M. Serafini, N. Suri, Efficient model checking of fault-tolerant distributed protocols, in: DSN, 2011, pp. 73–84.
[8] G. Bracha, S. Toueg, Asynchronous consensus and broadcast protocols, J. ACM 32 (4) (1985) 824–840.
[9] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (March 1996) 225–267.
[10] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an opensource tool for symbolic model checking, in: CAV, in: LNCS, vol. 2404, 2002, pp. 359–364.
[11] E.M. Clarke, D. Kroening, J. Ouaknine, O. Strichman, Completeness and complexity of bounded model checking, in: VMCAI, in: LNCS, vol. 2937, 2004, pp. 85–96.
[12] T.W. Doeppner, Parallel program correctness through refinement, in: POPL, 1977, pp. 155–169.
[13] P. Godefroid, Using partial orders to improve automatic verification methods, in: CAV, in: LNCS, vol. 531, 1990, pp. 176–185.
[14] A. John, I. Konnov, U. Schmid, H. Veith, J. Widder, Parameterized model checking of fault-tolerant distributed algorithms by abstraction, in: FMCAD, 2013, pp. 201–209.
[15] A. John, I. Konnov, U. Schmid, H. Veith, J. Widder, Towards modeling and model checking fault-tolerant distributed algorithms, in: SPIN, in: LNCS, vol. 7976, 2013, pp. 209–226.
[16] I. Konnov, ByMC: Byzantine model checker, http://forsyte.tuwien.ac.at/software/bymc/, 2015, accessed: Dec. 1.
[17] I. Konnov, H. Veith, J. Widder, SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms, in: CAV (Part I), in: LNCS, vol. 9206, 2015, pp. 85–102.
[18] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear completeness thresholds for bounded model checking, in: CAV, in: LNCS, vol. 6806, 2011, pp. 557–572.
[19] D. Kroening, O. Strichman, Efficient computation of recurrence diameters, in: VMCAI, in: LNCS, vol. 2575, 2003, pp. 298–309.
[20] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.

[21] L. Lamport, F.B. Schneider, Pretending atomicity, Tech. rep. 44, SRC, 1989.
[22] J. Leroux, G. Sutre, Flat counter automata almost everywhere!, in: ATVA, in: LNCS, vol. 3707, 2005, pp. 489–503.
[23] R.J. Lipton, Reduction: a method of proving properties of parallel programs, Commun. ACM 18 (12) (1975) 717–721.
[24] B.D. Lubachevsky, An approach to automating the verification of compact parallel coordination programs, II, Tech. rep. 64, New York University, Computer Science Department, 1983.
[25] B.D. Lubachevsky, An approach to automating the verification of compact parallel coordination programs. I, Acta Inform. 21 (2) (1984) 125–169.
[26] N. Lynch, Distributed Algorithms, Morgan, Kaufman, 1996.
[27] A. Mostéfaoui, E. Mourgaya, P.R. Parvédy, M. Raynal, Evaluating the condition-based approach to solve consensus, in: DSN, 2003, pp. 541–550.
[28] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.
[29] D. Peled, All from one, one for all: on model checking using representatives, in: CAV, in: LNCS, vol. 697, 1993, pp. 409–423.
[30] A. Pnueli, J. Xu, L. Zuck, Liveness with $(0, 1, \infty)$-counter abstraction, in: CAV, in: LNCS, vol. 2404, Springer, 2002, pp. 93–111.
[31] M. Raynal, A case study of agreement problems in distributed systems: non-blocking atomic commitment, in: HASE, 1997, pp. 209–214.
[32] T. Srikanth, S. Toueg, Simulating authenticated broadcasts to derive simple fault-tolerant algorithms, Distrib. Comput. 2 (1987) 80–94.
[33] A. Valmari, Stubborn sets for reduced state space generation, in: Advances in Petri Nets 1990, in: LNCS, vol. 483, Springer, 1991, pp. 491–515.

Chapter 5

Para²: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms

Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. Para²: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. Journal of Formal Methods in System Design, vol. 51, no. 2, pp. 270-307, 2017.

DOI: `http://dx.doi.org/10.1007/s10703-017-0297-4`

Form Methods Syst Des (2017) 51:270–307
DOI 10.1007/s10703-017-0297-4

Para²: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms

Igor Konnov[1] · **Marijana Lazić**[1] · **Helmut Veith**[1] · **Josef Widder**[1]

Published online: 20 September 2017
© The Author(s) 2017. This article is an open access publication

Abstract Automatic verification of threshold-based fault-tolerant distributed algorithms (FTDA) is challenging: FTDAs have multiple parameters that are restricted by arithmetic conditions, the number of processes and faults is parameterized, and the algorithm code is parameterized due to conditions counting the number of received messages. Recently, we introduced a technique that first applies data and counter abstraction and then runs bounded model checking (BMC). Given an FTDA, our technique computes an upper bound on the diameter of the system. This makes BMC complete for reachability properties: it always finds a counterexample, if there is an actual error. To verify state-of-the-art FTDAs, further improvement is needed. In contrast to encoding bounded executions of a counter system over an abstract finite domain in SAT, in this paper, we encode bounded executions over integer counters in SMT. In addition, we introduce a new form of reduction that exploits acceleration

Supported by: the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403 and S11405), project PRAVDA (P27722), and Doctoral College LogiCS (W1255-N23); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103). This is an extended version of the paper "SMT and POR beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms" that appeared in CAV (Part I), volume 9206 of LNCS, pages 85–102, 2015.

Josef Widder
widder@forsyte.at
http://forsyte.at/widder

Igor Konnov
konnov@forsyte.at
http://forsyte.at/konnov

Marijana Lazić
lazic@forsyte.at
http://forsyte.at/lazic

Helmut Veith
veith@forsyte.at
http://forsyte.at/veith

[1] Institute of Information Systems E184/4, TU Wien (Vienna University of Technology), Favoritenstraße 9-11, 1040 Vienna, Austria

and the structure of the FTDAs. This aggressively prunes the execution space to be explored by the solver. In this way, we verified safety of seven FTDAs that were out of reach before.

Keywords Parameterized verification · Bounded model checking · Completeness · Partial orders in distributed systems · Reduction · Fault-tolerant distributed algorithms · Byzantine faults

1 Introduction

Replication is a classic approach to make computing systems more reliable. In order to avoid a single point of failure, one uses multiple processes in a distributed system. Then, if some of these processes fail (e.g., by crashing or deviating from their expected behavior) the distributed system as a whole should stay operational. For this purpose one uses fault-tolerant distributed algorithms (FTDAs). These algorithms have been extensively studied in distributed computing theory [1,50], and found application in safety critical systems (automotive or aeronautic industry). With the recent advent of data centers and cloud computing we observe growing interest in fault-tolerant distributed algorithms, and their correctness, also for more mainstream computer science applications [19,20,31,47,52,54,60].

We consider automatic verification techniques specifically for threshold-based fault-tolerant distributed algorithms. In these algorithms, processes collect messages from their peers, and check whether the number of received messages reaches a threshold, e.g., a threshold may ensure that acknowledgments from a majority of processes have been received. Waiting for majorities, or more generally waiting for quorums, is a key pattern of many fault-tolerant algorithms, e.g., consensus, replicated state machine, and atomic commit. In [34] we introduced an efficient encoding of these algorithms, which we used in [33] for abstraction-based parameterized model checking of safety and liveness of several case study algorithms, which are parameterized in the number of processes n and the fraction of faults t, e.g., $n > 3t$. In [41] we were able to verify reachability properties of more involved algorithms by applying bounded model checking. We showed how to make bounded model checking complete in the parameterized case. In particular, we considered counter systems where we record for each local state, how many processes are in this state. We have one counter per local state ℓ, denoted by $\kappa[\ell]$. A process step from local state ℓ to local state ℓ' is modeled by decrementing $\kappa[\ell]$ and incrementing $\kappa[\ell']$. When δ processes perform the same step one after the other, we allow the processes to do the *accelerated step* that instantaneously changes the two affected counters by δ. The number δ is called *acceleration factor*, which can vary in a single run.

As we focus on threshold-based FTDAs, we consider counter systems defined by *threshold automata*. Here, transitions are guarded by *threshold guards* that compare a shared integer variable to a linear combination of parameters, e.g., $x \geq n - t$ or $x < t$, where x is a shared variable and n and t are parameters.

Completeness of the method [41] with respect to reachability is shown by proving a bound on the diameter of the accelerated system. Inspired by Lamport's view of distributed computation as partial order on events [43], our method uses a reduction similar to Lipton's [48]. Instead of pruning executions that are "similar" to ones explored before as in *partial order reduction* [28,53,59], we use the partial order to show (offline) that every run has a similar run of bounded length. Interestingly, the bound is independent of the parameters. In [41], we introduced the following automated method, which combines this idea with data abstraction [33]:

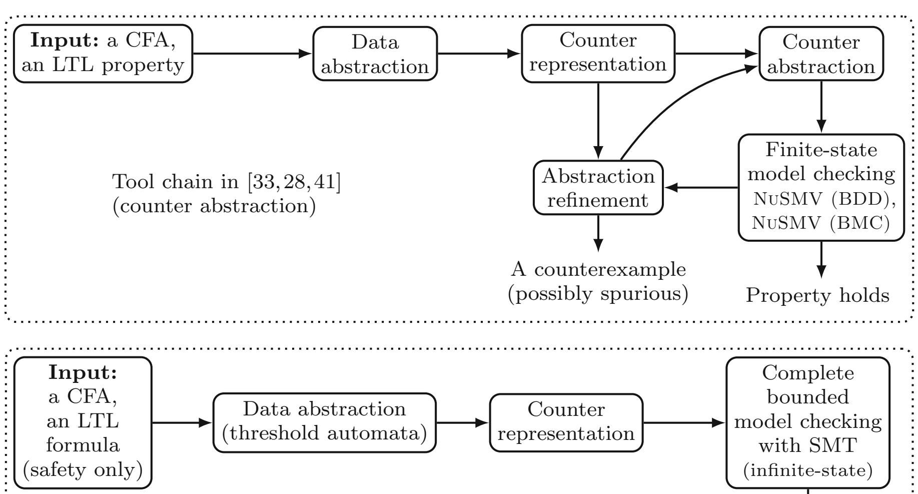


Fig. 1 Tool chain with counter abstraction [27,33,41] on top, and with SMT-based bounded model checking on bottom

1. Apply a parametric data abstraction to the process code to get a finite state process description, and construct the threshold automaton (TA) [33,36].
2. Compute the diameter bound, based on the control flow of the TA.
3. Construct a system with abstract counters, i.e., a counter abstraction [33,55].
4. Perform SAT-based bounded model checking [6,16] up to the diameter bound, to check whether bad states are reached in the counter abstraction.
5. If a counterexample is found, check its feasibility and refine, if needed [13,33].

Figure 1 gives on top a diagram [40] that shows the technique based on counter abstraction. While this allowed us to automatically verify several FTDAs not verified before, there remained two bottlenecks for scalability to larger and more complex protocols: First, counter abstraction can lead to spurious counterexamples. As counters range over a finite abstract domain, the semantics of abstract increment and decrements on the counters introduce nondeterminism. For instance, the value of a counter can remain unchanged after applying an increment. Intuitively, processes or messages can be "added" or "lost", which results in that, e.g., in the abstract model the number of messages sent is smaller than the number of processes that have sent a message, which obviously is spurious behavior. Second, counter abstraction works well in practice only for processes with a few dozens of local states. It has been observed in [4] that counter abstraction does not scale to hundreds of local states. We had similar experience with counter abstraction in our experiments in [41]. We conjecture that this is partly due to the many different interleavings, which result in a large search space.

To address these bottlenecks, we make two crucial *contributions* in this paper:

1. To eliminate one of the two sources of spurious counterexamples, namely, the nondeterminism added by abstract counters, we do bounded model checking using SMT solvers with linear integer arithmetic on the accelerated system, instead of SAT-based bounded model checking on the counter abstraction.

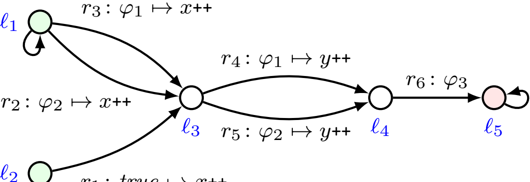


Fig. 2 An example threshold automaton with threshold guards "$\varphi_1: x \geq \lceil (n+t)/2 \rceil - f$", "$\varphi_2: y \geq (t+1) - f$", and "$\varphi_3: y \geq (2t+1) - f$"

2. We reduce the search space dramatically: we introduce the notion of an *execution schema* that is defined as a sequence of local rules of the TA. By assigning to each rule of a schema an acceleration factor (possibly 0, which models that no process executes the rule), one obtains a run of the counter system. Hence, due to parameterization, each schema represents infinitely many runs. We show how to construct a set of schemas whose set of reachable states coincides with the set of reachable states of the accelerated counter system.

The resulting method is depicted at the bottom of Fig. 1. Our construction can be seen as an aggressive form of reduction, where each run has a similar run generated by a schema from the set. To show this, we capture the guards that are locked and unlocked in a *context*. Our key insight is that a bounded number of transitions changes the context in each run. For example, of all transitions increasing a variable x, at most one makes $x \geq n - t$ true, and at most one makes $x < t + 1$ false (the parameters n and t are fixed in a run, and shared variables can only be increased). We fix those transitions that change the context, and apply the ideas of reduction to the subexecutions between these transitions.

Our experiments show that SMT solvers and schemas outperform SAT solvers and counter abstraction in parameterized verification of threshold-based FTDAs. We verified safety of FTDAs [10,18,29,51,56,57] that have not been automatically verified before. In addition we achieved dramatic speedup and reduced memory footprint for FTDAs [9,12,58] which previously were verified in [41].

In this article we focus on parameterized reachability properties. Recently, we extended this approach to safety and liveness, for which we used the reachability technique of this article as a black box [37].

2 Our approach at a glance

For modeling threshold-based FTDAs, we use threshold automata that were introduced in [38, 41] and are discussed in more detail in [40]. We use Fig. 2 to describe our contributions in this section. The figure presents a threshold automaton TA over two shared variables x and y and parameters n, t, and f, which is inspired by the distributed asynchronous broadcast protocol from [9]. There, $n - f$ correct processes concurrently follow the control flow of TA, and f processes are Byzantine faulty. As is typical for FTDAs, the parameters must satisfy a resilience condition, e.g., $n > 3t \wedge t \geq f \geq 0$, that is, less than a third of the processes are faulty. The circles depict the local states $\ell_1, \ldots, \ell_5$, two of them are the initial states ℓ_1 and ℓ_2. The edges depict the rules $r_1, \ldots, r_6$ labeled with guarded commands $\varphi \mapsto$ act, where φ is one of the threshold guards "$\varphi_1: x \geq \lceil (n+t)/2 \rceil - f$", "$\varphi_2: y \geq (t+1) - f$", and "$\varphi_3: y \geq (2t+1) - f$", and an action act increases the shared variables (x and y) by one, or zero (as in rule r_6).

We associate with every local state ℓ_i a non-negative counter $\boldsymbol{\kappa}[\ell_i]$ that represents the number of processes in ℓ_i. Together with the values of x, y, n, t, and f, the values of the counters constitute a *configuration* of the system. In the initial configuration there are $n - f$ processes in initial states, i.e., $\boldsymbol{\kappa}[\ell_1] + \boldsymbol{\kappa}[\ell_2] = n - f$, and the other counters and the shared variables x and y are zero.

The rules define the transitions of the counter system. For example, according to the rule r_2, if in the current configuration the guard $y \geq t + 1 - f$ holds true and $\boldsymbol{\kappa}[\ell_1] \geq 5$, then five processes can instantaneously move out of the local state ℓ_1 to the local state ℓ_3, and increment x as prescribed by the action of r_2 (since the evaluation of the guard $y \geq t + 1 - f$ cannot change from true to false). This results in increasing x and $\boldsymbol{\kappa}[\ell_3]$ by five, and decreasing the counter $\boldsymbol{\kappa}[\ell_1]$ by five. When, as in our example, rule r_2 is conceptually executed by 5 processes, we denote this transition by $(r_2, 5)$, where 5 is the acceleration factor. A sequence of transitions forms a *schedule*, e.g., $(r_1, 2)$, $(r_3, 1)$, $(r_1, 1)$.

In this paper, we address a parameterized reachability problem, e.g., can at least one correct process reach the local state ℓ_5, when $n - f$ correct processes start in the local state ℓ_1? Or, in terms of counter systems, is a configuration with $\boldsymbol{\kappa}[\ell_5] \neq 0$ reachable from an initial configuration with $\boldsymbol{\kappa}[\ell_1] = n - f \wedge \boldsymbol{\kappa}[\ell_2] = 0$? As discussed in [41], acceleration does not affect reachability, and precise treatment of the resilience condition and threshold guards is crucial for solving this problem.

2.1 Schemas

When applied to a configuration, a schedule generates a *path*, that is, an alternating sequence of configurations and transitions. As initially x and y are zero, threshold guards φ_1, φ_2, and φ_3 evaluate to false. As rules may increase variables, these guards may eventually become true. In our example we do not consider guards like $x < t + 1$ that are initially true and become false, although we formally treat them in our technique. In fact, initially only r_1 is unlocked. Because r_1 increases x, it may unlock φ_1. Thus r_4 becomes unlocked. Rule r_4 increases y and thus repeated application of r_4 (by different processes) first unlocks φ_2 and then φ_3. We introduce a notion of a *context* that is the set of threshold guards that evaluate to true in a configuration. For our example we observe that each path goes through the following sequence of contexts $\{\}$, $\{\varphi_1\}$, $\{\varphi_1, \varphi_2\}$, and $\{\varphi_1, \varphi_2, \varphi_3\}$. In fact, the sequence of contexts in a path is always monotonic, as the shared variables can only be increased.

The conjunction of the guards in the context $\{\varphi_1, \varphi_2\}$ implies the guards of the rules r_1, r_2, r_3, r_4, r_5; we call these rules unlocked in the context. This motivates our definition of a *schema*: a sequence of contexts and rules. We give an example of a schema below, where inside the curly brackets we give the contexts, and fixed sequences of rules in between. (We discuss the underlined rules below.)

$$\begin{aligned} S = \{\}\ &\underline{r_1}, \underline{r_1}\ \{\varphi_1\}\ \underline{r_1}, \underline{r_3}, r_4, \underline{r_4}\ \{\varphi_1, \varphi_2\} \\ &r_1, r_2, r_3, r_4, r_5, r_4, \underline{r_5}\ \{\varphi_1, \varphi_2, \varphi_3\}\ r_1, r_2, r_3, r_4, \underline{r_5}, \underline{r_6}\ \{\varphi_1, \varphi_2, \varphi_3\} \end{aligned} \tag{2.1}$$

Given a schema, we can generate a schedule by attaching to each rule an acceleration factor, which can possibly be 0. For instance, if we attach non-zero factors to the underlined rules in S, and a zero factor to the other rules, we generate the following schedule τ' (we omit the transitions with 0 factors here).

$$\tau' = \underbrace{(r_1, 1)}_{\tau'_1}, \underbrace{(r_1, 1)}_{t_1}, \underbrace{(r_1, 1), (r_3, 1)}_{\tau'_2}, \underbrace{(r_4, 1)}_{t_2}, \underbrace{}_{\tau'_3}\underbrace{(r_5, 1)}_{t_3}, \underbrace{(r_5, 2), (r_6, 4)}_{\tau'_4} \tag{2.2}$$

Springer

It can easily be checked that τ' is generated by schema S, because the sequence of the underlined rules in S matches the sequence of rules appearing in τ'.

In this paper, we show that the schedules generated by a few schemas—one per each monotonic sequence of contexts—span the set of all reachable configurations. To this end, we apply reduction and acceleration to relate arbitrary schedules to their representatives, which are generated by schemas.

2.2 Reduction and acceleration

In this section we show what we mean by a schedule being "related" to its representative. Consider, e.g., the following schedule τ from the initial state σ_0 with $n = 5$, $t = f = 1$, $\boldsymbol{\kappa}[\ell_1] = 1$, and $\boldsymbol{\kappa}[\ell_2] = 3$:

$$\tau = \underbracket{(r_1, 1),}_{\tau_1} \underbrace{(r_1, 1),}_{t_1} \underbracket{(r_3, 1), (r_1, 1),}_{\tau_2} \underbrace{(r_4, 1),}_{t_2} \underbracket{\quad}_{\tau_3} \underbrace{(r_5, 1),}_{t_3}$$
$$\underbracket{(r_6, 1), (r_5, 1), (r_5, 1), (r_6, 1), (r_6, 1), (r_6, 1)}_{\tau_4}$$

Observe that after $(r_1, 1), (r_1, 1)$, variable $x = 2$, and thus φ_1 is true. Hence transition t_1 changes the context from $\{\}$ to $\{\varphi_1\}$. Similarly t_2 and t_3 change the context. Context changing transitions are marked with curly brackets. Between them we have the subschedules $\tau_1, \ldots, \tau_4$ (τ_3 is empty) marked with square brackets.

To show that this schedule is captured by the schema (2.1), we apply partial order arguments—that is, a mover analysis [48]—regarding distributed computations: As the guards φ_2 and φ_3 evaluate to true in τ_4, and r_5 precedes r_6 in the control flow of the TA, all transitions $(r_5, 1)$ can be moved to the left in τ_4. Similarly, $(r_1, 1)$ can be moved to the left in τ_2. The resulting schedule is applicable and leads to the same configuration as the original one. Further, we can accelerate the adjacent transitions with the same rule, e.g., the sequence $(r_5, 1), (r_5, 1)$ can be transformed into $(r_5, 2)$. Thus, we transform subschedules τ_i into τ_i', and arrive at the schedule τ' from Eq. (2.2), which we call the representative schedule of τ. As the representative schedule τ' is generated from the schema in (2.1), we say that the schema captures schedule τ. (It also captures τ'.) Importantly for reachability checking, if τ and τ' are applied to the same configuration, they end in the same configuration. These arguments are formalized in Sects. 5, 6 and 7.

2.3 Encoding a schema in SMT

One of the key insights in this paper is that reachability checking via schemas can be encoded efficiently as SMT queries in linear integer arithmetic. In more detail, finite paths of counter systems can be expressed with inequalities over counters such as $\boldsymbol{\kappa}[\ell_2]$ and $\boldsymbol{\kappa}[\ell_3]$, shared variables such as x and y, parameters such as n, t, and f, and acceleration factors. Also, threshold guards and resilience conditions are expressions in linear integer arithmetic.

We give an example of reachability checking with SMT using the *simple* schema $\{\}\, r_1, r_1\, \{\varphi_1\}$ which is contained in the schema S in Eq. (2.1). To obtain a complete encoding for S, one can similarly encode the other simple schemas and combine them.

To this end, we have to express constraints on three configurations σ_0, σ_1, and σ_2. For the initial configuration σ_0, we introduce integer variables: $\boldsymbol{\kappa}_1^0, \ldots, \boldsymbol{\kappa}_5^0$ for local state counters, x^0 and y^0 for shared variables, and n, t, and f for parameters. As is written in Eq. (2.3), the configuration σ_0 should satisfy the initial constraints, and its context should be empty (that is, all guards evaluate to false):

$$\boldsymbol{\kappa}_1^0 + \boldsymbol{\kappa}_2^0 = n - f \wedge \boldsymbol{\kappa}_3^0 = \boldsymbol{\kappa}_4^0 = \boldsymbol{\kappa}_5^0 = 0 \wedge x^0 = y^0 = 0$$
$$\wedge n \geq 3t \wedge t \geq f \geq 0 \wedge (\neg\varphi_1 \wedge \neg\varphi_2 \wedge \neg\varphi_3)[x^0/x, y^0/y] \quad (2.3)$$

The configuration σ_1 is reached from σ_0 by applying a transition with the rule r_1 and an acceleration factor δ^1, and the configuration σ_2 is reached from σ_1 by applying a transition with the rule r_1 and an acceleration factor δ^2. Applying transition with the rule r_1 to σ_0 just means to increase both $\boldsymbol{\kappa}[\ell_3]$ and x by δ^1 and decrease $\boldsymbol{\kappa}[\ell_2]$ by δ^1. Hence, we introduce four fresh variables per transition and add the arithmetic operations. According to the schema, configuration σ_2 has the context $\{\varphi_2\}$. The following equations express these constraints[1]:

$$\boldsymbol{\kappa}_3^1 = \boldsymbol{\kappa}_3^0 + \delta^1 \wedge \boldsymbol{\kappa}_2^1 = \boldsymbol{\kappa}_2^0 - \delta^1 \wedge x^1 = x^0 + \delta^1 \quad (2.4)$$
$$\boldsymbol{\kappa}_3^2 = \boldsymbol{\kappa}_3^1 + \delta^2 \wedge \boldsymbol{\kappa}_2^2 = \boldsymbol{\kappa}_2^1 - \delta^2 \wedge x^2 = x^1 + \delta^2$$
$$\wedge(\varphi_1 \wedge \neg\varphi_2 \wedge \neg\varphi_3)[x^2/x, y^0/y] \quad (2.5)$$

Finally, we express the reachability question for all paths generated by the simple schema $\{\}\ r_1, r_1\ \{\varphi_1\}$. Whether there is a configuration with $\boldsymbol{\kappa}[\ell_5] \neq 0$ reachable from an initial configuration with $\boldsymbol{\kappa}[\ell_1] = n - f$ and $\boldsymbol{\kappa}[\ell_2] = 0$ can then be encoded as:

$$\boldsymbol{\kappa}_1^0 = n - f \wedge \boldsymbol{\kappa}_2^0 = 0 \wedge \boldsymbol{\kappa}_5^0 \neq 0 \quad (2.6)$$

Note that we check only $\boldsymbol{\kappa}_5^0$ against zero, as the local state ℓ_5 is never updated by the rule r_1. It is easy to see that conjunction of Eqs. (2.3)–(2.6) does not have a solution, and thus all paths generated by the schema $\{\}\ r_1, r_1\ \{\varphi_1\}$ do not reach a configuration with $\boldsymbol{\kappa}[\ell_5] \neq 0$. By writing down constraints for the other three simple schemas in Eq. (2.1), we can check reachability for the paths generated by the whole schema as well. As discussed in Sect. 2.1, our results also imply reachability on all paths whose representatives are generated by the schema. More details on the SMT encoding can be found in Sect. 9.

3 Parameterized counter systems

We recall the framework of [41] to the extent necessary, and extend it with the notion of a context in Sect. 3.2. A threshold automaton describes a process in a concurrent system, and is a tuple $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ defined below.

The finite set $\mathcal{L}$ contains the *local states*, and $\mathcal{I} \subseteq \mathcal{L}$ is the set of *initial states*. The finite set Γ contains the *shared variables* that range over the natural numbers $\mathbb{N}_0$. The finite set Π is a set of *parameter variables* that range over $\mathbb{N}_0$, and the *resilience condition RC* is a formula over parameter variables in linear integer arithmetic, e.g., $n > 3t$. The set of *admissible parameters* is $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : \mathbf{p} \models RC\}$.

A key ingredient of threshold automata are threshold guards (or, just guards):

Definition 3.1 A *threshold guard* is an inequality of one of the following two forms:

(R) $x \geq a_0 + a_1 \cdot p_1 + \cdots + a_{|\Pi|} \cdot p_{|\Pi|}$, or
(F) $x < a_0 + a_1 \cdot p_1 + \cdots + a_{|\Pi|} \cdot p_{|\Pi|}$,

where $x \in \Gamma$ is a shared variable, $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$ are integer coefficients, and $p_1, \ldots, p_{|\Pi|} \in \Pi$ are parameters. We denote the set of all guards of the form (R) by Φ^{rise}, and the set of all guards of the form (F) by Φ^{fall}.

[1] Our model requires *all* variables to be non-negative integers. Although these constraints (e.g., $x^1 \geq 0$) have to be encoded in the SMT queries, we omit these constraints here for a more concise presentation.

A rule defines a conditional transition between local states that may update the shared variables. Formally, a *rule* is a tuple $(from, to, \varphi^{\text{rise}}, \varphi^{\text{fall}}, \mathbf{u})$: the local states *from* and *to* are from $\mathcal{L}$. (Intuitively, they capture from which local state to which a process moves.) A rule is only executed if the conditions φ^{rise} and φ^{fall} evaluate to true. Condition φ^{rise} is a conjunction of guards from Φ^{rise}, and φ^{fall} is a conjunction of guards from Φ^{fall} (cf. Definition 3.1). We denote the set of guards used in φ^{rise} by $\mathsf{guard}(\varphi^{\text{rise}})$, and $\mathsf{guard}(\varphi^{\text{fall}})$ is the set of guards used in φ^{fall}.

Rules may increase shared variables using an update vector $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$ that is added to the vector of shared variables. As $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$, global variables can only be increased or left unchanged. As will be later formalized in Proposition 3.1, guards from Φ^{rise} can only change from false to true (rise), and guards from Φ^{fall} can change from true to false (fall). Finally, $\mathcal{R}$ is the finite set of rules. We use the dot notation to refer to components of rules, e.g., $r.from$ or $r.\mathbf{u}$.

Example 3.1 In Fig. 2, the rule $r_2\colon \varphi_2 \mapsto x{+}{+}$ that describes a transition from ℓ_1 to ℓ_3, can formally be written as $(\ell_1, \ell_3, \varphi_2, \top, (1, 0))$. Its intuitive meaning is as follows. If the guard $\varphi_2 : y \geq (t+1) - f$ evaluates to true, a process can move from the local state ℓ_1 to the local state ℓ_3, and the global variable x is incremented, while y remains unchanged. We formalize the semantics as counter systems in Sect. 3.1.

Definition 3.2 Given a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *precedence relation* $\prec_P$: for a pair of rules $r_1, r_2 \in \mathcal{R}$, it holds that $r_1 \prec_P r_2$ if and only if $r_1.to = r_2.from$. We denote by $\prec_P^+$ the transitive closure of $\prec_P$. Further, we say that $r_1 \sim_P r_2$, if $r_1 \prec_P^+ r_2 \wedge r_2 \prec_P^+ r_1$, or $r_1 = r_2$.

Assumption 3.3 We limit ourselves to threshold automata relevant for FTDAs, i.e., those where $r.\mathbf{u} = \mathbf{0}$ for all rules $r \in \mathcal{R}$ that satisfy $r \prec_P^+ r$. Such automata were called *canonical* in [41].

Remark 3.1 We use threshold automata to model fault-tolerant distributed algorithms that count messages from distinct senders. These algorithms are based on an "idealistic" reliable communication assumption (no message loss); these assumptions are typically expected to be ensured by "lower level bookkeeping code", e.g., communication protocols. As a result, the algorithms we consider here do not gain from sending the same message (that is, increasing a variable) inside a loop, so that we can focus on threshold automata that do not increase shared variables in loops.

Example 3.2 In the threshold automaton from Fig. 3 we have that $r_2 \prec_P r_3 \prec_P r_4 \prec_P r_5 \prec_P r_6 \prec_P r_8 \prec_P r_2$. Thus, we have that $r_2 \prec_P^+ r_2$. In our case this implies that $r_2.\mathbf{u} = \mathbf{0}$ by definition. Similarly we can conclude that $r_3.\mathbf{u} = r_4.\mathbf{u} = r_5.\mathbf{u} = r_6.\mathbf{u} = r_7.\mathbf{u} = r_8.\mathbf{u} = \mathbf{0}$.

Looplets The relation $\sim_P$ defines equivalence classes of rules. An equivalence class corresponds to a loop or to a single rule that is not part of a loop. Hence, we use the term looplet for one such equivalence class. For a given set of rules $\mathcal{R}$ let $\mathcal{R}/\!\sim$ be the set of equivalence classes defined by $\sim_P$. We denote by $[r]$ the equivalence class of rule r. For two classes c_1 and c_2 from $\mathcal{R}/\!\sim$ we write $c_1 \prec_C c_2$ iff there are two rules r_1 and r_2 in $\mathcal{R}$ satisfying $[r_1] = c_1$ and $[r_2] = c_2$ and $r_1 \prec_P^+ r_2$ and $r_1 \not\sim_P r_2$. As the relation $\prec_C$ is a strict partial order, there are linear extensions of $\prec_C$. Below, we fix an *arbitrary* of these linear extensions to sort transitions in a schedule: We denote by $\prec_C^{lin}$ a linear extension of $\prec_C$.

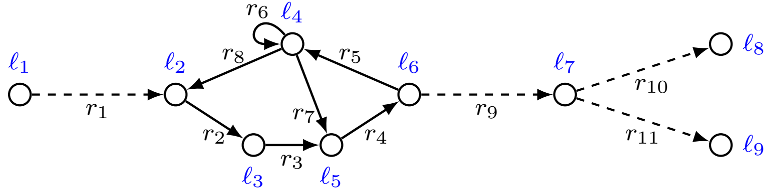


Fig. 3 A threshold automaton TA with local states $\mathcal{L} = \{\ell_i : 1 \le i \le 9\}$ and rules $\mathcal{R} = \{r_i : 1 \le i \le 11\}$. The rules drawn with solid arrows $\{r_2, \ldots, r_8\}$ constitute a single equivalence class, while all other rules are singleton equivalence classes

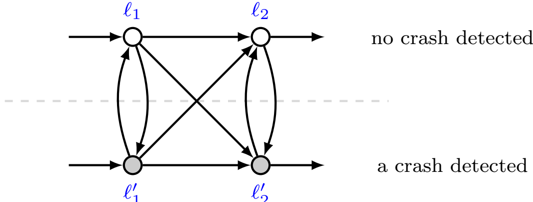


Fig. 4 A typical structure found in threshold automata that model fault-tolerant algorithms with a failure detector [12]. The gray circles depict those local states, where the failure detector reports a crash. The local states ℓ_i and ℓ'_i differ only in the output of the failure detector. As the failure detector output changes non-deterministically, the threshold automaton contains loops of size two

Example 3.3 Consider Fig. 3. The threshold automaton has five looplets: $c_1 = \{r_1\}$, $c_2 = \{r_2, \ldots, r_8\}$, $c_3 = \{r_9\}$, $c_4 = \{r_{10}\}$, and $c_5 = \{r_{11}\}$. From $r_9 \prec_P r_{10}$, it follows that $c_3 \prec_C c_4$, and from $r_4 \prec_P^+ r_{10}$, it follows that $c_2 \prec_C c_4$. We can pick two linear extensions of $\prec_C$, denoted by $\prec_1$ and $\prec_2$. We have $c_1 \prec_1 \cdots \prec_1 c_5$, and $c_1 \prec_2 c_2 \prec_2 c_3 \prec_2 c_5 \prec_2 c_4$. In this paper we always fix one linear extension.

Remark 3.2 It may seem natural to collapse such loops into singleton local states. In our case studies, e.g, [29], non-trivial loops are used to express non-deterministic choice due to failure detectors [12], as shown in Fig. 4. Importantly, some local states inside the loops appear in the specifications. Thus, one would have to use arguments from distributed computing to characterize when collapsing states is sound. In this paper, we present a technique that deals with the loops without need for additional modelling arguments.

3.1 Counter systems

We use a function $N : \mathbf{P}_{RC} \to \mathbb{N}_0$ to capture the number of processes for each combination of parameters. As we use SMT, we assume that N can be expressed in linear integer arithmetic. For instance, if only correct processes are explictly modeled we typically have $N(n, t, f) = n - f$, and the respective SMT expression is $n - f$. Given N, a threshold automaton TA, and admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, we define a counter system as a transition system (Σ, I, R). It consists of the set of configurations Σ, which contain evaluations of the counters and variables, the set of initial configurations I, and the transition relation R:

Configurations Σ and I A configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ consists of a vector of *counter values* $\sigma.\boldsymbol{\kappa} \in \mathbb{N}_0^{|\mathcal{L}|}$ (for simplicity we use the convention that $\mathcal{L} = \{1, \ldots, |\mathcal{L}|\}$) a vector of *shared variable values* $\sigma.\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$, and a vector of *parameter values* $\sigma.\mathbf{p} = \mathbf{p}$. The set Σ is the

set of all configurations. The set of initial configurations I contains the configurations that satisfy $\sigma.\mathbf{g} = \mathbf{0}$, $\sum_{i \in \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = N(\mathbf{p})$, and $\sum_{i \notin \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = 0$. This means that in every initial configuration all global variables have zero values, and all $N(\mathbf{p})$ modeled processes are located only in the initial local states.

Example 3.4 Consider the threshold automaton from Fig. 2 with the initial states ℓ_1 and ℓ_2. Let us consider a system of five processes, one of them being Byzantine faulty. Thus, $n = 5$, $t = f = 1$, and we explicitely model $N(5, 1, 1) = n - f = 4$ correct processes. One of the initial configurations is $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$, where $\sigma.\boldsymbol{\kappa} = (1, 3, 0, 0, 0)$, $\sigma.\mathbf{g} = (0, 0)$, and $\sigma.\mathbf{p} = (5, 1, 1)$. In other words, there is one process in ℓ_1, three processes in ℓ_2, and global variables are initially $x = y = 0$. Note that $\sum_{i \in \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = \boldsymbol{\kappa}[\ell_1] + \boldsymbol{\kappa}[\ell_2] = 1 + 3 = 4 = N(5, 1, 1)$.

Transition relation R A *transition* is a pair $t = (rule, factor)$ of a rule of the TA and a non-negative integer called the *acceleration factor*, or just factor for short. (As already discussed in Sect. 2.1, we will use the zero factors when generating schedules from schemas.) For a transition $t = (rule, factor)$ we refer by $t.\mathbf{u}$ to $rule.\mathbf{u}$, and by $t.\varphi^{\text{fall}}$ to $rule.\varphi^{\text{fall}}$, etc. We say a transition t is *unlocked* in configuration σ if $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + k \cdot t.\mathbf{u}, \sigma.\mathbf{p}) \models t.\varphi^{\text{rise}} \wedge t.\varphi^{\text{fall}}$, for $k \in \{0, \ldots, t.factor - 1\}$. Note that here we use a notation that a configuration satisfies a formula, which is considered true if and only if the formula becomes true when all free variables of the formulas are evaluated as in the configuration.

We say that transition t is *applicable (or enabled)* in configuration σ, if it is unlocked, and $\sigma.\boldsymbol{\kappa}[t.from] \geq t.factor$. (As all counters are non-negative, a transition with the zero *factor* is always applicable to all configurations provided that the guards are unlocked.) We say that σ' is the result of applying the enabled transition t to σ, and write $\sigma' = t(\sigma)$, if

- $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.factor \cdot t.\mathbf{u}$ and $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
- if $t.from \neq t.to$ then
  - $\sigma'.\boldsymbol{\kappa}[t.from] = \sigma.\boldsymbol{\kappa}[t.from] - t.factor$,
  - $\sigma'.\boldsymbol{\kappa}[t.to] = \sigma.\boldsymbol{\kappa}[t.to] + t.factor$, and
  - $\forall \ell \in \mathcal{L} \setminus \{t.from, t.to\}$ it holds that $\sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$
- if $t.from = t.to$ then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$

The transition relation $R \subseteq \Sigma \times \Sigma$ of the counter system is defined as follows: $(\sigma, \sigma') \in R$ iff there is a rule $r \in \mathcal{R}$ and a factor $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$. Updates do not decrease the values of shared variables, and thus the following proposition was introduced in [41]:

Proposition 3.1 [41] *For all configurations σ, all rules r, and all transitions t applicable to σ, the following holds:*

1. *If $\sigma \models r.\varphi^{\text{rise}}$ then $t(\sigma) \models r.\varphi^{\text{rise}}$*
2. *If $t(\sigma) \not\models r.\varphi^{\text{rise}}$ then $\sigma \not\models r.\varphi^{\text{rise}}$*
3. *If $\sigma \not\models r.\varphi^{\text{fall}}$ then $t(\sigma) \not\models r.\varphi^{\text{fall}}$*
4. *If $t(\sigma) \models r.\varphi^{\text{fall}}$ then $\sigma \models r.\varphi^{\text{fall}}$*

Schedules and paths A *schedule* is a (finite) sequence of transitions. For a schedule τ and an index $i : 1 \leq i \leq |\tau|$, by $\tau[i]$ we denote the ith transition of τ, and by τ^i we denote the prefix $\tau[1], \ldots, \tau[i]$ of τ. A schedule $\tau = t_1, \ldots, t_m$ is *applicable* to configuration σ_0, if there is a sequence of configurations $\sigma_1, \ldots, \sigma_m$ with $\sigma_i = t_i(\sigma_{i-1})$ for $1 \leq i \leq m$. If there is a $t_i.factor > 1$, then a schedule is *accelerated*.

By $\tau \cdot \tau'$ we denote the concatenation of two schedules τ and τ'. A sequence $\sigma_0, t_1, \sigma_1, \ldots, \sigma_{k-1}, t_k, \sigma_k$ of alternating configurations and transitions is called a (finite)

path, if transition t_i is enabled in σ_{i-1} and $\sigma_i = t_i(\sigma_{i-1})$, for $1 \le i \le k$. For a configuration σ_0 and a schedule τ applicable to σ_0, by $\mathsf{path}(\sigma_0, \tau)$ we denote the path $\sigma_0, t_1, \ldots, t_{|\tau|}, \sigma_{|\tau|}$ with $t_i = \tau[i]$ and $\sigma_i = t_i(\sigma_{i-1})$, for $1 \le i \le |\tau|$.

3.2 Contexts and slices

The evaluation of the guards in the sets Φ^{rise} and Φ^{fall} in a configuration solely defines whether certain transitions are unlocked (but not necessarily enabled). From Proposition 3.1, one can see that after a transition has been applied, more guards from Φ^{rise} may get unlocked and more guards from Φ^{fall} may get locked. In other words, more guards from Φ^{rise} may evaluate to true and more guards from Φ^{fall} may evaluate to false. To capture this intuition, we define:

Definition 3.4 A *context* Ω is a pair $(\Omega^{\text{rise}}, \Omega^{\text{fall}})$ with $\Omega^{\text{rise}} \subseteq \Phi^{\text{rise}}$ and $\Omega^{\text{fall}} \subseteq \Phi^{\text{fall}}$. We denote by $|\Omega| = |\Omega^{\text{rise}}| + |\Omega^{\text{fall}}|$.

For two contexts $(\Omega_1^{\text{rise}}, \Omega_1^{\text{fall}})$ and $(\Omega_2^{\text{rise}}, \Omega_2^{\text{fall}})$, we define that $(\Omega_1^{\text{rise}}, \Omega_1^{\text{fall}}) \sqsubset (\Omega_2^{\text{rise}}, \Omega_2^{\text{fall}})$ if and only if $\Omega_1^{\text{rise}} \cup \Omega_1^{\text{fall}} \subset \Omega_2^{\text{rise}} \cup \Omega_2^{\text{fall}}$. Then, a sequence of contexts $\Omega_1, \ldots, \Omega_m$ is *monotonically increasing*, if $\Omega_i \sqsubset \Omega_{i+1}$, for $1 \le i < m$. Further, a monotonically increasing sequence of contexts $\Omega_1, \ldots, \Omega_m$ is *maximal*, if $\Omega_1 = (\emptyset, \emptyset)$ and $\Omega_m = (\Phi^{\text{rise}}, \Phi^{\text{fall}})$ and $|\Omega_{i+1}| = |\Omega_i| + 1$, for $1 \le i < m$. We obtain:

Proposition 3.2 *Every maximal monotonically increasing sequence of contexts is of length* $|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}| + 1$. *There are at most* $(|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}|)!$ *such sequences.*

Example 3.5 For the example in Fig. 2, we have $\Phi^{\text{rise}} = \{\varphi_1, \varphi_2, \varphi_3\}$, and $\Phi^{\text{fall}} = \emptyset$. Thus, there are $(|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}|)! = 6$ maximal monotonically increasing sequences of contexts. Two of them are $(\emptyset, \emptyset) \sqsubset (\{\varphi_1\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_2\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_2, \varphi_3\}, \emptyset)$ and $(\emptyset, \emptyset) \sqsubset (\{\varphi_3\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_3\}, \emptyset) \sqsubset (\{\varphi_1, \varphi_2, \varphi_3\}, \emptyset)$. All of them have length $|\Phi^{\text{rise}}| + |\Phi^{\text{fall}}| + 1 = 4$.

To every configuration σ, we attach the context consisting of all guards in Φ^{rise} that evaluate to true in σ, and all guards in Φ^{fall} that evaluate to *false* in σ:

Definition 3.5 Given a threshold automaton, we define its *configuration context* as a function $\omega : \Sigma \to 2^{\Phi^{\text{rise}}} \times 2^{\Phi^{\text{fall}}}$ that for each configuration $\sigma \in \Sigma$ gives a context $(\Omega^{\text{rise}}, \Omega^{\text{fall}})$ with $\Omega^{\text{rise}} = \{\varphi \in \Phi^{\text{rise}} : \sigma \models \varphi\}$ and $\Omega^{\text{fall}} = \{\varphi \in \Phi^{\text{fall}} : \sigma \not\models \varphi\}$.

The following monotonicity result is a direct consequence of Proposition 3.1.

Proposition 3.3 *If a transition t is enabled in a configuration* σ, *then either* $\omega(\sigma) \sqsubset \omega(t(\sigma))$, *or* $\omega(\sigma) = \omega(t(\sigma))$.

Definition 3.6 A schedule τ is *steady* for a configuration σ, if for every prefix τ' of τ, the context does not change, i.e., $\omega(\tau'(\sigma)) = \omega(\sigma)$.

Proposition 3.4 *A schedule* τ *is steady for a configuration* σ *if and only if* $\omega(\sigma) = \omega(\tau(\sigma))$.

In the following definition, we associate a sequence of contexts with a path:

Definition 3.7 Given a configuration σ and a schedule τ applicable to σ, we say that $\mathsf{path}(\sigma, \tau)$ *is consistent with* a sequence of contexts $\Omega_1, \ldots, \Omega_m$, if there exist indices $n_0, \ldots, n_m$, with $0 = n_0 \le n_1 \le \ldots \le n_m = |\tau| + 1$, such that for every k, $1 \le k \le m$, and every i with $n_{k-1} \le i < n_k$, it holds that $\omega(\tau^i(\sigma)) = \Omega_k$.

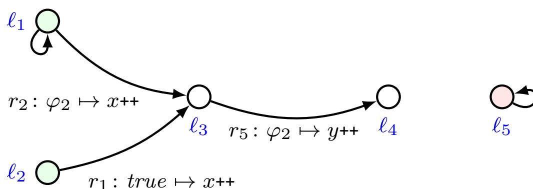


Fig. 5 The slice of the TA in Fig. 2 that is constructed for the context $(\{\varphi_2\}, \emptyset)$

Every path is consistent with a uniquely defined maximal monotonically increasing sequence of contexts. (Some of the indices $n_0, \dots, n_m$ in Definition 3.7 may be equal.) In Sect. 4, we use such sequences of contexts to construct a schema recognizing many paths that are consistent with the same sequence of contexts.

A context defines which rules of the TA are unlocked. A schedule that is steady for a configuration visits only one context, and thus we can statically remove TA's rules that are locked in the context:

Definition 3.8 Given a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ and a context Ω, we define the *slice* of TA with context $\Omega = (\Omega^{\text{rise}}, \Omega^{\text{fall}})$ as a threshold automaton $\mathsf{TA}|_\Omega = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}|_\Omega, RC)$, where a rule $r \in \mathcal{R}$ belongs to $\mathcal{R}|_\Omega$ if and only if $\left(\bigwedge_{\varphi \in \Omega^{\text{rise}}} \varphi\right) \rightarrow r.\varphi^{\text{rise}}$ and $\left(\bigwedge_{\psi \in \Phi^{\text{fall}} \setminus \Omega^{\text{fall}}} \psi\right) \rightarrow r.\varphi^{\text{fall}}$.

In other words, $\mathcal{R}|_\Omega$ contains those and only those rules r with guards that evaluate to true in all configurations σ with $\omega(\sigma) = \Omega$. These are exactly the guards from $\Omega^{\text{rise}} \cup (\Phi^{\text{fall}} \setminus \Omega^{\text{fall}})$. When $\omega(\sigma) = \Omega$, then all guards from Ω^{rise} evaluate to true, and then $r.\varphi^{\text{rise}}$ must also be true. As Ω^{fall} contains those guards from Φ^{fall} that evaluate to false in σ, then all other guards from Φ^{fall} must evaluate to true, and then $r.\varphi^{\text{fall}}$ must be true too. Figure 5 shows an example of a slice.

3.3 Model checking problem: parameterized reachability

Given a threshold automaton TA, a *state property* B is a Boolean combination of formulas that have the form $\bigwedge_{i \in Y} \boldsymbol{\kappa}[i] = 0$, for some $Y \subseteq \mathcal{L}$. The *parameterized reachability* problem is to decide whether there are parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$, with $\sigma_0.\mathbf{p} = \mathbf{p}$, and a schedule τ, such that τ is applicable to σ_0, and property B holds in the final state: $\tau(\sigma_0) \models B$.

4 Main result: a complete set of schemas

To address parameterized reachability, we introduce schemas, i.e., alternating sequences of contexts and rule sequences. A schema serves as a pattern for a set of paths, and is used to efficiently encode parameterized reachability in SMT. As parameters give rise to infinitely many initial states, a schema captures an *infinite* set of paths. We show how to construct a *finite* set of schemas $\mathcal{S}$ with the following property: for each schedule τ and each configuration σ there is a representative schedule $s(\tau)$ such that: (1) applying $s(\tau)$ to σ results in $\tau(\sigma)$, and (2) $\mathsf{path}(\sigma, s(\tau))$ is generated by a schema from $\mathcal{S}$.

Definition 4.1 A schema is a sequence $\Omega_0, \rho_1, \Omega_1, \dots, \rho_m, \Omega_m$ of alternating contexts and rule sequences. We often write $\{\Omega_0\}\rho_1\{\Omega_1\}\dots\{\Omega_{m-1}\}\rho_m\{\Omega_m\}$ for a schema. A schema with two contexts is called simple.

Given two schemas $S_1 = \Omega_0, \rho_1, \ldots, \rho_k, \Omega_k$ and $S_2 = \Omega'_0, \rho'_1, \ldots, \rho'_m, \Omega'_m$ with $\Omega_k = \Omega'_0$, we define their *composition* $S_1 \circ S_2$ to be the schema that is obtained by concatenation of the two sequences: $\Omega_0, \rho_1, \ldots, \rho_k, \Omega'_0, \rho'_1, \ldots, \rho'_m, \Omega'_m$.

Definition 4.2 Given a configuration σ and a schedule τ applicable to σ, we say that $\mathsf{path}(\sigma, \tau)$ is generated by a simple schema $\{\Omega\}\,\rho\,\{\Omega'\}$, if the following hold:

- For $\rho = r_1, \ldots, r_k$ there is a monotonically increasing sequence of indices $i(1), \ldots, i(m)$, i.e., $1 \le i(1) < \cdots < i(m) \le k$, and there are factors $f_1, \ldots, f_m \ge 0$, so that schedule $(r_{i(1)}, f_1), \ldots, (r_{i(m)}, f_m) = \tau$.
- The first and the last states match the contexts: $\omega(\sigma) = \Omega$ and $\omega(\tau(\sigma)) = \Omega'$.

In general, we say that $\mathsf{path}(\sigma, \tau)$ is generated by a schema S, if $S = S_1 \circ \cdots \circ S_k$ for simple schemas $S_1, \ldots, S_k$ and $\tau = \tau_1 \cdots \tau_k$ such that each $\mathsf{path}(\pi_i(\sigma), \tau_i)$ is generated by the simple schema S_i, for $\pi_i = \tau_1 \cdots \tau_{i-1}$ and $1 \le i \le k$.

Remark 4.1 Definition 4.2 allows schemas to generate paths that have transitions with zero acceleration factors. Applying a transition with a zero factor to a configuration σ results in the same configuration σ, which corresponds to a stuttering step. This does not affect reachability. In the following, we will apply Definition 4.2 to representative paths that may have transitions with zero factors.

Example 4.1 Let us go back to the example of a schema S and a schedule τ' introduced in Eqs. (2.1) and (2.2) in Sect. 2.1. It is easy to see that schema S can be decomposed into four simple schemas $S_1 \circ \cdots \circ S_4$, e.g., $S_1 = \{\}\, r_1, r_1\, \{\varphi_1\}$ and $S_2 = \{\varphi_1\}\, r_1, r_3, r_4, r_4\, \{\varphi_1, \varphi_2\}$. Consider an initial state σ_0 with $n = 5$, $t = f = 1$, $x = y = 0$, $\boldsymbol{\kappa}[\ell_1] = 1$, $\boldsymbol{\kappa}[\ell_2] = 3$, and $\boldsymbol{\kappa}[\ell_i] = 0$ for $i \in \{3, 4, 5\}$. To ensure that $\mathsf{path}(\sigma_0, \tau')$ is generated by schema S, one has to check Definition 4.2 for schemas $S_1, \ldots, S_4$ and schedules $(\tau'_1 \cdot t_1)$, $(\tau'_2 \cdot t_2)$, $(\tau'_3 \cdot t_3)$, and τ'_4, respectively. For instance, $\mathsf{path}(\sigma_0, \tau'_1 \cdot t_1)$ is generated by S_1. Indeed, take the sequence of indices 1 and 2 and the sequence of acceleration factors 1 and 1. The path $\mathsf{path}(\sigma_0, \tau'_1 \cdot t_1)$ ends in the configuration σ_1 that differs from σ_0 in that $\boldsymbol{\kappa}[\ell_2] = 1$, $\boldsymbol{\kappa}[\ell_3] = 2$, and $x = 2$. The contexts $\omega(\sigma_0) = (\{\}, \{\})$ and $\omega(\sigma_1) = (\{\varphi_1\}, \{\})$ match the contexts of schema S_1, as required by Definition 4.2.

Similarly, $\mathsf{path}(\sigma_1, \tau'_2 \cdot t_2)$ is generated by schema S_2. To see that, compare the contexts and use the index sequence 1, 2, 4, and acceleration factors 1.

The *language of a schema* S—denoted with $\mathcal{L}(S)$—is the set of all paths generated by S. For a set of configurations $C \subseteq \Sigma$ and a set of schemas $\mathcal{S}$, we define the set $\mathsf{Reach}(C, \mathcal{S})$ to contain all configurations reachable from C via the paths generated by the schemas from $\mathcal{S}$, i.e., $\mathsf{Reach}(C, \mathcal{S}) = \{\tau(\sigma) \mid \sigma \in C,\ \exists S \in \mathcal{S}.\ \mathsf{path}(\sigma, \tau) \in \mathcal{L}(S)\}$. We say that a set $\mathcal{S}$ of schemas is *complete*, if for every set of configurations $C \subseteq \Sigma$ it is the case that the set of all states reachable from C via the paths generated by the schemas from $\mathcal{S}$, is exactly the set of all possible states reachable from C. Formally, $\forall C \subseteq \Sigma.\ \{\tau(\sigma) \mid \sigma \in C,\ \tau \text{ is applicable to } \sigma\} = \mathsf{Reach}(C, \mathcal{S})$.

In [41], a quantity $\mathcal{C}$ has been introduced that depends on the number of conditions in a TA. It has been shown that for every configuration σ and every schedule τ applicable to σ, there is a schedule τ' of length at most $d = |\mathcal{R}| \cdot (\mathcal{C} + 1) + \mathcal{C}$ that is also applicable to σ and results in $\tau(\sigma)$ [41, Thm. 8]. Hence, by enumerating all sequences of rules of length up to d, one can construct a complete set of schemas:

Theorem 4.1 *For a threshold automaton, there is a complete schema set $\mathcal{S}_d$ of cardinality $|\mathcal{R}|^{|\mathcal{R}|\cdot(\mathcal{C}+1)+\mathcal{C}}$.*

Although the set $\mathcal{S}_d$ is finite, enumerating all its elements is impractical. We show that there is a complete set of schemas whose cardinality solely depends on the number of guards that syntactically occur in the TA. These numbers $|\Phi^{\mathrm{rise}}|$ and $|\Phi^{\mathrm{fall}}|$ are in practice much smaller than the number of rules $|\mathcal{R}|$:

Theorem 4.2 *For all threshold automata, there exists a complete schema set of cardinality at most* $(|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$. *In this set, the length of each schema does not exceed* $(3 \cdot |\Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}| + 2) \cdot |\mathcal{R}|$.

In the following sections we prove the ingredients of the following argument for the theorem: construct the set Z of all maximal monotonically increasing sequences of contexts. From Proposition 3.2, we know that there are at most $(|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$ maximal monotonically increasing sequences of contexts. Therefore, $|Z| \le (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$. Then, for each sequence $z \in Z$, we do the following:

(1) We show that for each configuration σ and each schedule τ applicable to σ and consistent with the sequence z, there is a schedule $s(\tau)$ that has a specific structure, and is also applicable to σ. We call $s(\tau)$ the representative of τ. We introduce and formally define this specific structure of representative schedules in Sects. 5, 6 and 7. We prove existence and properties of the representative schedule in Theorem 7.1 (Sect. 7). Before that we consider special cases: when all rules of a schedule belong to the same looplet (Theorem 5.1, Sect. 5), and when a schedule is steady (Theorem 6.1, Sect. 6).
(2) Next we construct a schema (for the sequence z) and show that it generates all paths of all schedules $s(\tau)$ found in (1). The length of the schema is at most $(3 \cdot (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|) + 2) \cdot |\mathcal{R}|$. This is shown in Theorem 7.2 (Sect. 7).

Theorem 4.2 follows from the above theorems, which we prove in the following.

Remark 4.2 Let us stress the difference between [41] and this work. From [41], it follows that in order to check correctness of a TA it is sufficient to check only the schedules of bounded length $d(\mathsf{TA})$. The bound $d(\mathsf{TA})$ does not depend on the parameters, and can be computed for each TA. The proofs in [41] demonstrate that every schedule longer than $d(\mathsf{TA})$ can be transformed into an "equivalent" representative schedule, whose length is bounded by $d(\mathsf{TA})$. Consequently, one can treat every schedule of length up to $d(\mathsf{TA})$ as its own representative schedule. Similar reasoning does not apply to the schemas constructed in this paper: (i) we construct a complete set of schemas, whose cardinality is substantially smaller than $|\mathcal{S}_d|$, and (ii) the schemas constructed in this paper can be twice as long as the schemas in $\mathcal{S}_d$.

As discussed in Remark 3.2, the looplets in our case studies are typically either singleton looplets or looplets of size two. In fact, most of our benchmarks have singleton looplets only, and thus their threshold automata can be reduced to directed acyclic graphs. The theoretical constructs of Sect. 5.2 are presented for the more general case of looplets of any size. For most of the benchmarks—the ones not using failure detectors—we need only the simple construction laid out in Sect. 5.1.

5 Case I: one context and one looplet

We show that for each schedule that uses only the rules from a fixed looplet and does not change its context, there exists a representative schedule of bounded length that reaches the

same final state. The goal is to construct a single schema per looplet. The technical challenge is that this *single* schema must generate representative schedules for *all* possible schedules, where, intuitively, processes may move arbitrarily between all local states in the looplet. As a consequence, the rules that appear in the representative schedules can differ from the rules that appear in the arbitrary schedules visiting a looplet.

We fix a threshold automaton, a context Ω, a configuration σ with $\omega(\sigma) = \Omega$, a looplet c, and a schedule τ applicable to σ and using only rules from c. We then construct the representative schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ and the schema $\mathsf{cschema}_c^{\Omega}$.

The technical details of the construction of $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ for the case when $|c| = 1$ is given in Sect. 5.1, and for the case when $|c| > 1$ in Sect. 5.2. We show in Sect. 5.3 that these constructions give us a schedule that has the desired properties: it reaches the same final state as the given schedule τ, and its length does not exceed $2 \cdot |c|$.

Note that in [41], the length of the representative schedule was bounded by $|c|$. However, all representative schedules of a looplet in this section can be generated by a single looplet schema.

5.1 Singleton looplet

Let us consider the case of the looplet c containing only one transition, that is, $|c| = 1$. There is a trivial representative schedule of a single transition:

Lemma 5.1 *Given a threshold automaton, a configuration σ, and a schedule $\tau = (r, f_1), \dots, (r, f_m)$ applicable to σ, one of the two schedules is also applicable to σ and results in $\tau(\sigma)$: schedule $(r, f_1 + \dots + f_m)$, or schedule $(r, 0)$.*

Proof We distinguish two cases:

Case $r.to = r.from$ Then, $r.\mathbf{u} = \mathbf{0}$, and $\tau^k(\sigma) = \sigma$ for $0 \le k \le |\tau|$. Consequently, the schedule $(r, 0)$ is applicable to σ, and it results in $\tau(\sigma) = \sigma$.

Case $r.to \ne r.from$ We prove by induction on the length $k : 1 \le k \le m$ of a prefix of τ, that the following constraints hold for all k:

$$(\tau^k(\sigma)).\boldsymbol{\kappa}[r.from] = \sigma.\boldsymbol{\kappa}[r.from] - (f_1 + \dots + f_k) \tag{5.1}$$

$$(\tau^k(\sigma)).\mathbf{g} = \sigma.\mathbf{g} + (f_1 + \dots + f_k) \cdot r.\mathbf{u} \tag{5.2}$$

$$(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\mathrm{fall}} \wedge r.\varphi^{\mathrm{rise}} \text{ for all } f \in \{0, \dots, f_1 + \dots + f_k\} \tag{5.3}$$

Base case $k = 1$. As schedule τ is applicable to σ, its first transition is enabled in σ. Thus, by the definition of an enabled transition, the rule r is unlocked, i.e., for all $f \in \{0, \dots, f_1\}$, it holds $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + f_1 \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\mathrm{fall}} \wedge r.\varphi^{\mathrm{rise}}$. By the definition, once the transition $\tau[1]$ is applied, it holds that $\tau^1(\sigma).\boldsymbol{\kappa}[from] = \sigma.\boldsymbol{\kappa}[from] - f_1$ and $(\tau^k(\sigma)).\mathbf{g} = \sigma.\mathbf{g} + f_1 \cdot r.\mathbf{u}$. Thus, Constraints (5.1)–(5.3) are satisfied for $k = 1$.

Inductive step $k > 1$. As schedule τ is applicable to σ, its prefix τ^k is applicable to σ. Hence, transition $\tau[k]$ is applicable to $\tau^{k-1}(\sigma)$.

By the definition of an enabled transition, for all $f \in \{0, \dots, f_k\}$, it holds

$$((\tau^{k-1}(\sigma)).\boldsymbol{\kappa}, ((\tau^{k-1}(\sigma)).\mathbf{g} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\mathrm{fall}} \wedge r.\varphi^{\mathrm{rise}}.$$

By applying the Eq. (5.2) for $k - 1$ of the inductive hypothesis, we obtain that for all $f \in \{0, \dots, f_k\}$, it holds that $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + (f_1 + \dots + f_{k-1} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\mathrm{fall}} \wedge r.\varphi^{\mathrm{rise}}$. By combining this constraint with the constraint (5.3) for $k - 1$, we arrive at the constraint (5.3) for k.

By applying $\tau[k]$, we get that $(\tau^k(\sigma)).\boldsymbol{\kappa}[r.\mathit{from}] = (\tau^{k-1}(\sigma)).\boldsymbol{\kappa}[r.\mathit{from}] - f_k$ and $(\tau^k(\sigma)).\mathbf{g} = (\tau^{k-1}(\sigma)).\mathbf{g} + f_k \cdot r.\mathbf{u}$. By applying (5.1) and (5.2) for $k-1$ to these equations, we arrive at the Eqs. (5.1) and (5.2) for k.

Based on (5.1) and (5.3) for all values of k, and in particular $k = m$, we can now show applicability. From Eq. (5.1), we immediately obtain that $\sigma.\boldsymbol{\kappa}[r.\mathit{from}] \geq f_1 + \cdots + f_m$. From constraint (5.3), we obtain that $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + f \cdot r.\mathbf{u}, \sigma.\mathbf{p}) \models r.\varphi^{\text{fall}} \wedge r.\varphi^{\text{rise}}$ for all $f \in \{0, \ldots, f_1 + \cdots + f_m\}$. These are the required conditions for the transition $(r, f_1 + \cdots + f_m)$ to be applicable to the configuration σ. □

Consequently, when c has a single rule r, for configuration σ and a schedule $\tau = (r, f_1), \ldots, (r, f_m)$, Lemma 5.1 allows us to take the singleton schedule (r, f) as $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ and to take the singleton schema $\{\Omega\}\, r\, \{\Omega\}$ as $\mathsf{cschema}_c^{\Omega}$. The factor f is either $f_1 + \ldots + f_m$ or zero.

5.2 Non-singleton looplet

Next we focus on non-singleton looplets. Thus, we assume that $|c| > 1$. Our construction is based on two directed trees, whose undirected versions are spanning trees, sharing the same root. In order to find a representative of a steady schedule τ which leads from σ to $\tau(\sigma)$, we determine for each local state how many processes have to move in or out of the state, and then we move them along the edges of the trees. First, we give the definitions of such trees, and then we show how to use them to construct the representative schedules and the schema.

Spanning out-trees and in-trees We construct the *underlying graph of looplet* c, that is, a directed graph G_c, whose vertices consist of local states that appear as components *from* or *to* of the rules from c, and the edges are the rules of c. More precisely, we construct a directed graph $G_c = (V_c, E_c, L_c)$, whose edges from E_c are labeled by function $L_c : E_c \to c$ with the rules of c as follows:

$$V_c = \{\ell \mid \exists r \in c,\ r.\mathit{to} = \ell \vee r.\mathit{from} = \ell\},$$
$$E_c = \{(\ell, \ell') \mid \exists r \in c,\ r.\mathit{from} = \ell,\ r.\mathit{to} = \ell'\},$$
$$L_c((\ell, \ell')) = r, \text{ if } r.\mathit{from} = \ell,\ r.\mathit{to} = \ell' \text{ for } (\ell, \ell') \in E_c \text{ and } r \in c.$$

Lemma 5.2 *Given a threshold automaton and a non-singleton looplet $c \in \mathcal{R}/{\sim}$, graph G_c is non-empty and strongly connected.*

Proof As, $|c| > 1$ and thus $E_c \geq 2$, graph G_c is non-empty. To prove that G_c is strongly connected, we consider a pair of rules $r_1, r_2 \in c$. By the definition of a looplet, it holds that $r_1 \prec_P^+ r_2$ and $r_2 \prec_P^+ r_1$. Thus, there is a path in G_c from $r_1.\mathit{to}$ to $r_2.\mathit{from}$, and there is a path in G_c from $r_2.\mathit{to}$ to $r_1.\mathit{from}$. As r_1 and r_2 correspond to some edges in G_c, there is a cycle that contains the vertices $r_1.\mathit{from}$, $r_1.\mathit{to}$, $r_2.\mathit{from}$, and $r_2.\mathit{to}$. Thus, graph G_c is strongly connected. □

As G_c is non-empty and strongly connected, we can fix an arbitrary node $h \in V_c$—called a *hub*—and construct two directed trees, whose undirected versions are spanning trees of the undirected version of G_c. These are two subgraphs of G_c: a directed tree $T_{\mathsf{out}} = (V_c, E_{\mathsf{out}})$, whose edges $E_{\mathsf{out}} \subseteq E_c$ are *pointing away from* h (out-tree); a directed tree $T_{\mathsf{in}} = (V_c, E_{\mathsf{in}})$, whose edges $E_{\mathsf{in}} \subseteq E_c$ are *pointing to* h (in-tree). For every node $v \in V_c \backslash \{h\}$, it holds that $|\{u : (u, v) \in E_{\mathsf{out}}\}| = 1$ and $|\{w : (v, w) \in E_{\mathsf{in}}\}| = 1$.

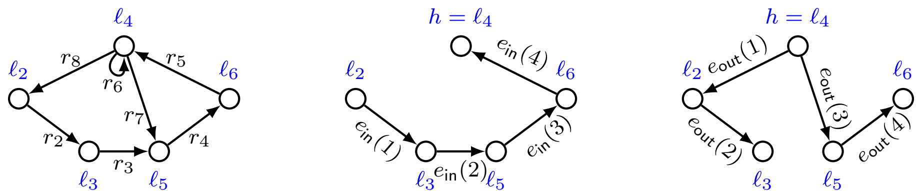


Fig. 6 The underlying graph of the looplet c_2 of the threshold automaton from Example 3.3 and Fig. 3 (left), together with trees T_{in} (middle) and T_{out} (right)

Further, we fix a topological order $\preceq_{\mathsf{in}}$ on the edges of tree T_{in}. More precisely, $\preceq_{\mathsf{in}}$ is such a partial order on E_{in} that for each pair of adjacent edges $(\ell, \ell'), (\ell', \ell'') \in E_{\mathsf{in}}$, it holds that $(\ell, \ell') \preceq_{\mathsf{in}} (\ell', \ell'')$. In the same way, we fix a topological order $\preceq_{\mathsf{out}}$ on the edges of tree T_{out}.

Example 5.1 Consider again the threshold automaton from Example 3.3 and Fig. 3. We construct trees T_{in} and T_{out} for looplet c_2, shown in Fig. 6.

Note that $V_c = \{\ell_2, \ell_3, \ell_4, \ell_5, \ell_6\}$, and $E_c = \{(\ell_2, \ell_3), (\ell_3, \ell_5), (\ell_5, \ell_6), (\ell_6, \ell_4), (\ell_4, \ell_4), (\ell_4, \ell_5), (\ell_4, \ell_2)\}$. Fix ℓ_4 as a hub. We can fix a linear order $\preceq_{\mathsf{in}}$ such that $(\ell_2, \ell_3) \preceq_{\mathsf{in}} (\ell_3, \ell_5) \preceq_{\mathsf{in}} (\ell_5, \ell_6) \preceq_{\mathsf{in}} (\ell_6, \ell_4)$, and a linear order $\preceq_{\mathsf{out}}$ such that $(\ell_4, \ell_2) \preceq_{\mathsf{out}} (\ell_2, \ell_3) \preceq_{\mathsf{out}} (\ell_4, \ell_5) \preceq_{\mathsf{out}} (\ell_5, \ell_6)$.

Note that for the chosen hub l_4 and this specific example, T_{in} and $\preceq_{\mathsf{in}}$ are uniquely defined, while an out-tree can be different from T_{out} from our Fig. 6 (the rules r_8, r_2, r_3, r_4 constitute a different tree from the same hub). Because out-tree T_{out} is not a chain, several linear orders different from $\preceq_{\mathsf{out}}$ can be chosen, e.g., $(\ell_4, \ell_2) \preceq_{\mathsf{out}} (\ell_4, \ell_5) \preceq_{\mathsf{out}} (\ell_2, \ell_3) \preceq_{\mathsf{out}} (\ell_5, \ell_6)$.

Representatives of non-singleton looplets Using these trees, we show how to construct a representative $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ of a schedule τ applicable to σ with $\sigma' = \tau(\sigma)$. For a configuration σ and a schedule τ applicable to σ, consider the trees T_{in} and T_{out}. We construct two sequences: the sequence $e_{\mathsf{in}}(1), \ldots, e_{\mathsf{in}}(|E_{\mathsf{in}}|)$ of all edges of T_{in} following the order $\preceq_{\mathsf{in}}$, i.e., if $e_{\mathsf{in}}(i) \preceq_{\mathsf{in}} e_{\mathsf{in}}(j)$, then $i \le j$; the sequence $e_{\mathsf{out}}(1), \ldots, e_{\mathsf{out}}(|E_{\mathsf{out}}|)$ of all edges of T_{out} following the order $\preceq_{\mathsf{out}}$. Further, we define the sequence of rules $r_{\mathsf{in}}(1), \ldots, r_{\mathsf{in}}(|E_{\mathsf{in}}|)$ with $r_{\mathsf{in}}(i) = L_c(e_{\mathsf{in}}(i))$ for $1 \le i \le |E_{\mathsf{in}}|$, and the sequence of rules $r_{\mathsf{out}}(1), \ldots, r_{\mathsf{out}}(|E_{\mathsf{out}}|)$ with $r_{\mathsf{out}}(i) = L_c(e_{\mathsf{out}}(i))$ for $1 \le i \le |E_{\mathsf{out}}|$. Using configurations σ and $\sigma' = \tau(\sigma)$, we define:

$$\delta_{\mathsf{in}}(i) = \sigma.\boldsymbol{\kappa}[f] - \sigma'.\boldsymbol{\kappa}[f], \text{ for } f = r_{\mathsf{in}}(i).\mathit{from} \text{ and } 1 \le i \le |E_{\mathsf{in}}|,$$
$$\delta_{\mathsf{out}}(j) = \sigma'.\boldsymbol{\kappa}[t] - \sigma.\boldsymbol{\kappa}[t], \text{ for } t = r_{\mathsf{out}}(j).\mathit{to} \text{ and } 1 \le j \le |E_{\mathsf{out}}|.$$

If $\delta_{\mathsf{in}}(i) \ge 0$, then $\delta_{\mathsf{in}}(i)$ processes should leave the local state $r_{\mathsf{in}}(i).\mathit{from}$ towards the hub, and they do it exclusively using the edge $e_{\mathsf{in}}(i)$. If $\delta_{\mathsf{out}}(j) \ge 0$, then $\delta_{\mathsf{out}}(j)$ processes should reach the state $r_{\mathsf{out}}(j).\mathit{to}$ from the hub, and they do it exclusively using the edge $e_{\mathsf{out}}(j)$. The negative values of $\delta_{\mathsf{in}}(i)$ and $\delta_{\mathsf{out}}(j)$ do not play any role in our construction, and thus, we use $\max(\delta_{\mathsf{in}}(i), 0)$ and $\max(\delta_{\mathsf{out}}(j), 0)$.

The main idea of the representative construction is as follows. First, we fire the sequence of rules $r_{\mathsf{in}}(1), \ldots, r_{\mathsf{in}}(k)$ to collect sufficiently many processes in the hub. Then, we fire the sequence of rules $r_{\mathsf{out}}(1), \ldots, r_{\mathsf{out}}(k)$ to distribute the required number of processes from the hub. As a result, for each location ℓ in the graph, the processes are transferred from ℓ

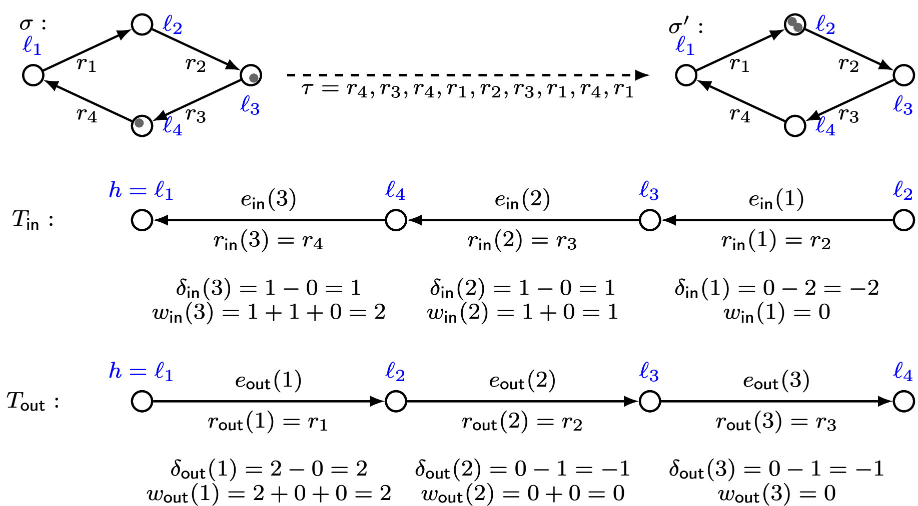


Fig. 7 Construction of the representative of a schedule using the rules in the four-element looplet, following Example 5.2

to the other locations, if $\sigma[\ell] > \sigma'[\ell]$, and additional processes arrive at ℓ, if $\sigma[\ell] < \sigma'[\ell]$. Using $\delta_{\mathsf{in}}(i)$ and $\delta_{\mathsf{out}}(i)$, we define the acceleration factors for each rule as follows:

$$w_{\mathsf{in}}(i) = \sum_{j\,:\,e_{\mathsf{in}}(j) \preceq_{\mathsf{in}} e_{\mathsf{in}}(i)} \max(\delta_{\mathsf{in}}(j), 0) \text{ and}$$

$$w_{\mathsf{out}}(i) = \sum_{j\,:\,e_{\mathsf{out}}(i) \preceq_{\mathsf{out}} e_{\mathsf{out}}(j)} \max(\delta_{\mathsf{out}}(j), 0).$$

Finally, we construct the schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ as follows:

$$\begin{aligned} \mathsf{crep}_c^{\Omega}[\sigma, \tau] = {} & (r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \ldots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|)), \\ & (r_{\mathsf{out}}(1), w_{\mathsf{out}}(1)), \ldots, (r_{\mathsf{out}}(|E_{\mathsf{out}}|), w_{\mathsf{out}}(|E_{\mathsf{out}}|)). \end{aligned} \tag{5.4}$$

Example 5.2 Consider the TA shown in Fig. 7. Let c be the four-element looplet that contains the rules r_1, r_2, r_3, and r_4, and τ be the schedule $\tau = (r_4, 1), (r_3, 1), (r_4, 1), (r_1, 1), (r_2, 1), (r_3, 1), (r_1, 1), (r_4, 1), (r_1, 1)$ that uses the rules of the looplet c. Consider a configuration σ with $\sigma.\boldsymbol{\kappa}[\ell_3] = \sigma.\boldsymbol{\kappa}[\ell_4] = 1$, and $\sigma.\boldsymbol{\kappa}[\ell_1] = \sigma.\boldsymbol{\kappa}[\ell_2] = 0$. The final configuration $\sigma' = \tau(\sigma)$ has the following properties: $\sigma'.\boldsymbol{\kappa}[\ell_2] = 2$ and $\sigma'.\boldsymbol{\kappa}[\ell_1] = \sigma'.\boldsymbol{\kappa}[\ell_3] = \sigma'.\boldsymbol{\kappa}[\ell_4] = 0$. By comparing σ and σ', we notice that one process should move from ℓ_3 to ℓ_2, and one from ℓ_4 to ℓ_2. We will now show how this is achieved by our construction.

For constructing the representative schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$, we first define trees T_{in} and T_{out}. If we chose ℓ_1 to be the hub, we get that $E_{\mathsf{in}} = \{(\ell_4, \ell_1), (\ell_3, \ell_4), (\ell_2, \ell_3)\}$, and thus the order is $(\ell_2, \ell_3) \preceq_{\mathsf{in}} (\ell_3, \ell_4) \preceq_{\mathsf{in}} (\ell_4, \ell_1)$. Therefore, we obtain $e_{\mathsf{in}}(1) = (\ell_2, \ell_3)$, $e_{\mathsf{in}}(2) = (\ell_3, \ell_4)$ and $e_{\mathsf{in}}(3) = (\ell_4, \ell_1)$. By calculating $\delta_{\mathsf{in}}(i)$ for every $i \in \{1, 2, 3\}$, we see that $\delta_{\mathsf{in}}(2) = 1$ and $\delta_{\mathsf{in}}(3) = 1$ are positive. Consequently, two processes go to the hub: one from $r_{\mathsf{in}}(2).\mathit{from} = \ell_3$ and one from $r_{\mathsf{in}}(3).\mathit{from} = \ell_4$. The coefficients w_{in} give us acceleration factors for all rules.

Similarly, we obtain $E_{\mathsf{out}} = \{(\ell_1, \ell_2), (\ell_2, \ell_3), (\ell_3, \ell_4)\}$, and the order must be $(\ell_1, \ell_2) \preceq_{\mathsf{out}} (\ell_2, \ell_3) \preceq_{\mathsf{out}} (\ell_3, \ell_4)$. Thus, $e_{\mathsf{out}}(1) = (\ell_1, \ell_2)$, $e_{\mathsf{in}}(2) = (\ell_2, \ell_3)$, and

$e_{\mathsf{out}}(3) = (\ell_3, \ell_4)$. Here only $\delta_{\mathsf{out}}(1) = 2$ has a positive value, and hence, two processes should move from hub to the local state $r_{\mathsf{out}}(1).to = \ell_2$. To achieve this, the acceleration factor of every rule $r_{\mathsf{out}}(i)$, $1 \le i \le 3$, must be $w_{\mathsf{out}}(i)$.

Therefore, by Eq. (5.4), the representative schedule is

$$\mathsf{crep}_c^{\Omega}[\sigma, \tau] = (r_2, 0), (r_3, 1), (r_4, 2), (r_1, 2), (r_2, 0), (r_3, 0).$$

Choosing another hub gives us another representative. For each hub, the representative is not longer than $2|c| = 8$, and leads to σ' when applied to σ.

In the following, we fix a threshold automaton TA, a context Ω, and a non-singleton looplet c of the slice $\mathsf{TA}|_{\Omega}$. We also fix a configuration σ of TA and a schedule τ that is contained in c and is applicable to σ. Our goal is to prove Lemma 5.8, which states that $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ is indeed applicable to σ and ends in $\tau(\sigma)$. To this end, we first prove auxiliary Lemmas 5.3–5.7.

Lemma 5.3 *For every* $i : 1 \le i \le |E_{\mathsf{in}}|$, *it holds that* $\sigma.\boldsymbol{\kappa}[r_i.from] \ge \max(\delta_{\mathsf{in}}(i), 0)$, *where* $r_i = L_c(e_{\mathsf{in}}(i))$.

Proof Recall that by the definition of a configuration, every counter $\sigma.\boldsymbol{\kappa}[\ell]$ is non-negative. If $\delta_{\mathsf{in}}(i) \ge 0$, then $\max(\delta_{\mathsf{in}}(i), 0) = \delta_{\mathsf{in}}(i) = \sigma.\boldsymbol{\kappa}[r_i.from] - \sigma'.\boldsymbol{\kappa}[r_i.from]$, which is bound from above by $\sigma.\boldsymbol{\kappa}[r_i.from]$. Otherwise, $\delta_{\mathsf{in}}(i) \le 0$, and we trivially have $\max(\delta_{\mathsf{in}}(i), 0) = 0$ and $0 \le \sigma.\boldsymbol{\kappa}[r_i.from]$. □

Lemma 5.4 *Schedule* $\tau_{\mathsf{in}} = (r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|))$ *is applicable to configuration* σ.

Proof We denote by α^i the schedule $(r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(i), w_{\mathsf{in}}(i))$, for $1 \le i \le |E_{\mathsf{in}}|$. Then $\tau_{\mathsf{in}} = \alpha^{|E_{\mathsf{in}}|}$.

All rules $r_{\mathsf{in}}(1), \dots, r_{\mathsf{in}}(|E_{\mathsf{in}}|)$ are from $\mathcal{R}|_{\Omega}$, and thus are unlocked. Hence, it is sufficient to show that the values of the locations from the set V_c are large enough to enable each transition $(r_{\mathsf{in}}(i), w_{\mathsf{in}}(i))$ for $1 \le i \le |E_{\mathsf{in}}|$. To this end, we prove by induction that $(\alpha^{i-1}(\sigma)).\boldsymbol{\kappa}[r_i.from] \ge w_{\mathsf{in}}(i)$, for $1 \le i \le |E_{\mathsf{in}}|$ and $r_i = L_c(e_{\mathsf{in}}(i))$.

Base case $i = 1$. For $r_1 = L_c(e_{\mathsf{in}}(1))$, we want to show that $\sigma.\boldsymbol{\kappa}[r_1.from] \ge w_{\mathsf{in}}(1)$. As $e_{\mathsf{in}}(1)$ is the first element of the sequence $e_{\mathsf{in}}(1), \dots, e_{\mathsf{in}}(E_{\mathsf{in}})$, which respects the order $\preceq_{\mathsf{in}}$, we conclude that $w_{\mathsf{in}}(1) = \max(\delta_{\mathsf{in}}(1), 0)$. From Lemma 5.3, it follows that $\sigma.\boldsymbol{\kappa}[r_1.from] \ge \max(\delta_{\mathsf{in}}(1), 0)$.

Inductive step k assume that for all $i : 1 \le i \le k-1 < |E_{\mathsf{in}}|$, schedule α^i is applicable to σ and show that $(\alpha^{k-1}(\sigma)).\boldsymbol{\kappa}[r_k.from] \ge w_{\mathsf{in}}(k)$ with $r_k = L_c(e_{\mathsf{in}}(k))$.

To this end, we construct the set of edges P_k that precede the edge $e_{\mathsf{in}}(k)$ in the topological order $\preceq_{\mathsf{in}}$, that is, $P_k = \{e \mid e \in E_{\mathsf{in}},\ e \preceq_{\mathsf{in}} e_{\mathsf{in}}(k),\ e \ne e_{\mathsf{in}}(k)\}$. We show that the following equation holds:

$$\alpha^{k-1}(\sigma)).\boldsymbol{\kappa}[r_k.from] = \sigma.\boldsymbol{\kappa}[r_k.from] + \sum_{e_{\mathsf{in}}(j) \in P_k} \max(\delta_{\mathsf{in}}(j), 0). \tag{5.5}$$

Indeed, if one picks an edge $e_{\mathsf{in}}(j) \in P_k$, the edge $e_{\mathsf{in}}(j)$ adds $w_{\mathsf{in}}(j)$ to the counter $\boldsymbol{\kappa}[r_k.from]$. As the sequence $\{e_{\mathsf{in}}(i)\}_{i \le k}$ is topologically sorted, it follows that $j < k$. Moreover, as the tree T_{in} is oriented towards the root, $e_{\mathsf{in}}(k)$ is the only edge leaving the local state $r_k.from$. Thus, no edge $e_{\mathsf{in}}(i)$ with $i < k$ decrements the counter $\sigma.\boldsymbol{\kappa}[r_k.from]$.

From Eq. (5.5) and Lemma 5.3, we conclude that $(\alpha^{k-1}(\sigma)).\boldsymbol{\kappa}[r_k.from]$ is not less than $\max(\delta_{\mathsf{in}}(k), 0) + \sum_{e_{\mathsf{in}}(j):\, e_{\mathsf{in}}(j) \preceq_{\mathsf{in}} e_{\mathsf{in}}(k),\ j \ne k} \max(\delta_{\mathsf{in}}(j), 0)$, which equals to $w_{\mathsf{in}}(k)$. This proves the inductive step.

Therefore, we have shown that $\tau_{\mathsf{in}} = \alpha^{|E_{\mathsf{in}}|}$ is applicable to σ. □

The following lemma is easy to prove by induction on the length of a schedule. The base case for a single transition follows from the definition of a counter system.

Lemma 5.5 *Let σ and σ' be two configurations and τ be a schedule applicable to σ such that $\tau(\sigma) = \sigma'$. Then it holds that $\sum_{\ell \in \mathcal{L}} (\sigma'[\ell] - \sigma[\ell]) = 0$.*

Further, we show that the required number of processes is reaching (or leaving) the hub, when the transitions derived from the trees T_{in} and T_{out} are executed:

Lemma 5.6 *The following equality holds:*

$$\sigma'.\boldsymbol{\kappa}[h] - \sigma.\boldsymbol{\kappa}[h] = \sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0) - \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0).$$

Proof Recall that T_{in} is a tree directed towards h, and the undirected version of T_{in} is a spanning tree of graph C. Hence, for each local state $\ell \in V_c \backslash \{h\}$, there is exactly one edge $e \in E_{\mathsf{in}}$ with $L_c(e).\mathit{from} = \ell$. Thus, the following equation holds:

$$\sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0) = \sum_{\ell \in V_c \backslash \{h\}} \max(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell], 0). \tag{5.6}$$

Similarly, T_{out} is a tree directed outwards h, and the undirected version of T_{out} is a spanning tree of graph C. Hence, for each local state $\ell \in V_c \backslash \{h\}$, there is exactly one edge $e \in E_{\mathsf{out}}$ with $L_c(e).\mathit{to} = \ell$. Thus, the following equation holds:

$$\sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0) = \sum_{\ell \in V_c \backslash \{h\}} \max(\sigma'.\boldsymbol{\kappa}[\ell] - \sigma.\boldsymbol{\kappa}[\ell], 0). \tag{5.7}$$

By combining (5.6) and (5.7), we obtain the following:

$$\begin{aligned}
&\sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0) - \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0) \\
&= \sum_{\ell \in V_c \backslash \{h\}} \big(\max(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell], 0) - \max(\sigma'.\boldsymbol{\kappa}[\ell] - \sigma.\boldsymbol{\kappa}[\ell], 0)\big) \\
&= \sum_{\ell \in V_c \backslash \{h\}} \big(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]\big) = \left(\sum_{\ell \in V_c} \sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]\right) - \big(\sigma.\boldsymbol{\kappa}[h] - \sigma'.\boldsymbol{\kappa}[h]\big).
\end{aligned} \tag{5.8}$$

As the initial schedule τ is applicable to σ, and $\tau(\sigma) = \sigma'$, by Lemma 5.5, $\sum_{\ell \in \mathcal{L}} (\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]) = 0$. As all rules in $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ are from $\mathcal{R}|_{\Omega}$ and thus change only the counters of local states in V_c, for each local state $\ell \in \mathcal{L} \backslash V_c$, its respective counter does not change, that is, $\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell] = 0$. Hence, $\sum_{\ell \in V_c} (\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell]) = 0$. From this and Eq. (5.8), the statement of the lemma follows. □

Lemma 5.7 *If τ_{in} denotes the schedule $(r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \ldots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|))$, the following equation holds:*

$$\tau_{\mathsf{in}}(\sigma).\boldsymbol{\kappa}[\ell] = \begin{cases} \sigma'.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0), & \text{if } \ell = h \\ \min(\sigma.\boldsymbol{\kappa}[\ell], \sigma'.\boldsymbol{\kappa}[\ell]), & \text{if } \ell \in V_c \backslash \{h\}. \end{cases}$$

Proof We prove the lemma by case distinction:

Case $\ell = h$ We show that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h] = \sigma.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0)$. Indeed, let P be the indices of edges coming into h, i.e., $P = \{i \mid 1 \le i \le |E_{\mathsf{in}}|,\ L_c(e_{\mathsf{in}}(i)) =$

$r,\ h = r.to\}$. As all edges in T_{in} are oriented towards h, it holds that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h]$ equals to $\sigma.\boldsymbol{\kappa}[h] + \sum_{i \in P} w_{\mathsf{in}}(i)$. By unfolding the definition of w_{in}, we obtain that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h] = \sigma.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{in}}|} \max(\delta_{\mathsf{in}}(i), 0)$. We observe that by Lemma 5.6, this sum equals to $\sigma'.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0)$. This proves the first case.

Case $\ell \in V_c \setminus \{h\}$ We show that $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[\ell] = \min(\sigma.\boldsymbol{\kappa}[\ell], \sigma'.\boldsymbol{\kappa}[\ell])$. Indeed, fix a node $\ell \in V_c \setminus \{h\}$ and construct two sets: the set of incoming edges $In = \{e_{\mathsf{in}}(i) \mid \exists \ell' \in V_c.\ e_{\mathsf{in}}(i) = (\ell', \ell)\}$ and the singleton set of outgoing edges $Out = \{e_{\mathsf{in}}(i) \mid \exists \ell' \in V_c.\ e_{\mathsf{in}}(i) = (\ell, \ell')\}$. By summing up the effect of all transitions in τ_{in}, we obtain $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell] + \sum_{e_{\mathsf{in}}(i) \in In} w_{\mathsf{in}}(i) - \sum_{e_{\mathsf{out}}(i) \in Out} w_{\mathsf{out}}(i)$. By unfolding the definition of w_{in}, we obtain $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell] - \sum_{e_{\mathsf{in}}(i) \in Out} \delta_{\mathsf{in}}(i)$, which can be rewritten as $\sigma.\boldsymbol{\kappa}[\ell] - \max(\sigma.\boldsymbol{\kappa}[\ell] - \sigma'.\boldsymbol{\kappa}[\ell], 0)$, which, in turn, equals to $\min(\sigma.\boldsymbol{\kappa}[\ell], \sigma'.\boldsymbol{\kappa}[\ell])$. This proves the second case. □

Now we are in a position to prove that schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ is applicable to configuration σ and results in configuration $\tau(\sigma)$:

Lemma 5.8 *The schedule* $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ *has the following properties: (a)* $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ *is applicable to* σ*, and (b)* $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ *results in* $\tau(\sigma)$ *when applied to* σ*.*

Proof Denote with τ_{in} the prefix $(r_{\mathsf{in}}(1), w_{\mathsf{in}}(1)), \dots, (r_{\mathsf{in}}(|E_{\mathsf{in}}|), w_{\mathsf{in}}(|E_{\mathsf{in}}|))$ of the schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$. For each $j : 1 \le j \le |E_{\mathsf{out}}|$, denote with β^j the prefix of $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ that has length of $|E_{\mathsf{in}}| + j$. Note that $\beta^{|E_{\mathsf{out}}|} = \mathsf{crep}_c^{\Omega}[\sigma, \tau]$.

Proving applicability of $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ *to* σ We notice that all rules in $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ are from $\mathcal{R}|_{\Omega}$ and thus are unlocked, and that τ_{in} is applicable to σ by Lemma 5.4. Hence, we only have to check that the values of counters from V_c are large enough, so that transitions $(r_{\mathsf{out}}(j), w_{\mathsf{out}}(j))$ can fire.

We prove that each schedule β^j is applicable to σ, for $j : 1 \le j \le |E_{\mathsf{out}}|$. We do so by induction on the distance from the root h in the tree T_{out}.

Base case root node h. Denote with O_h the set $\{(\ell, \ell') \in E_{\mathsf{out}} \mid \ell = h\}$. Let $j_1, \dots, j_m$ be the indices of all edges in O_h, and j_m be the maximum among them.

From Lemma 5.7, $(\tau_{\mathsf{in}}(\sigma)).\boldsymbol{\kappa}[h] = \sigma'.\boldsymbol{\kappa}[h] + \sum_{1 \le i \le |E_{\mathsf{out}}|} \max(\delta_{\mathsf{out}}(i), 0) = \sigma'.\boldsymbol{\kappa}[h] + \sum_{e_{\mathsf{out}}(j) \in O_h} w_{\mathsf{out}}(j)$. Thus, every transition $(e_{\mathsf{out}}(j), w_{\mathsf{out}}(j))$ with $e_{\mathsf{out}}(j) \in O_h$, is applicable to $\beta^{j-1}(\sigma)$. Also, $(\beta^{j_m}(\sigma)).\boldsymbol{\kappa}[h] = \sigma'.\boldsymbol{\kappa}[h]$.

Inductive step assume that for a node $\ell \in V_c$ and an edge $e_{\mathsf{out}}(k) = (\ell, \ell') \in E_{\mathsf{out}}$ outgoing from node ℓ, schedule β^k is applicable to configuration σ. Show that for each edge $e_{\mathsf{out}}(i)$ outgoing from node ℓ' the following hold: (i) schedule β^i is also applicable to σ; and (ii) $\beta^{|E_{\mathsf{out}}|}(\sigma).\boldsymbol{\kappa}[\ell'] = \sigma'.\boldsymbol{\kappa}[\ell']$.

(i) As the sequence $\{e_{\mathsf{out}}(j)\}_{j \le |E_{\mathsf{out}}|}$ is topologically sorted, for each edge $e_{\mathsf{out}}(i)$ outgoing from node ℓ', it holds that $k < i$.

From Lemma 5.7, we have that $\beta^k(\sigma).\boldsymbol{\kappa}[\ell'] = \min(\sigma.\boldsymbol{\kappa}[\ell'], \sigma'.\boldsymbol{\kappa}[\ell'])$. Because the transition $(e_{\mathsf{out}}(k), w_{\mathsf{out}}(k))$ adds $w_{\mathsf{out}}(k)$ to $\beta^{k-1}(\sigma).\boldsymbol{\kappa}[\ell']$, we have $\beta^k(\sigma).\boldsymbol{\kappa}[\ell'] = \min(\sigma.\boldsymbol{\kappa}[\ell'], \sigma'.\boldsymbol{\kappa}[\ell']) + w_{\mathsf{out}}(k)$. Let S be the set of all immediate successors of $e_{\mathsf{out}}(k)$, i.e., $S = \{i \mid \exists \ell''.\ (\ell', \ell'') = e_{\mathsf{out}}(i)\}$. From the definition of $w_{\mathsf{out}}(k)$, it follows that $w_{\mathsf{out}}(k) = \max(\delta_{\mathsf{out}}(k), 0) + \sum_{s \in S} w_{\mathsf{out}}(s)$. Thus, the transition $(e_{\mathsf{out}}(i), w_{\mathsf{out}}(i))$ for edge $e_{\mathsf{out}}(i)$ outgoing from node ℓ', can be executed.

(ii) Let $j_1, \dots, j_m$ be the indices of all edges outgoing from ℓ', and j_m be the maximum among them. From (i), it follows that

$$(\beta^{j_m}(\sigma)).\boldsymbol{\kappa}[\ell'] = \min(\sigma.\boldsymbol{\kappa}[\ell'], \sigma'.\boldsymbol{\kappa}[\ell']) + \max(\delta_{\mathsf{out}}(k), 0),$$

which equals to $\sigma'.\boldsymbol{\kappa}[\ell']$.

This proves that the schedule $\beta^{|E_{\mathsf{out}}|} = \mathsf{crep}_c^{\Omega}[\sigma, \tau]$ is applicable to σ.

Proving that $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ *results in* $\tau(\sigma)$ From the induction above, we conclude that for each $\ell \in V_c$, it holds that $(\beta^{|E_{\mathsf{out}}|}(\sigma)).\boldsymbol{\kappa}[\ell] = \sigma'.\boldsymbol{\kappa}[\ell]$. Edges in the trees T_{in} and T_{out} change only local states from V_c. We conclude that for all $\ell \in \mathcal{L}$, it holds that $\mathsf{crep}_c^{\Omega}[\sigma, \tau](\sigma).\boldsymbol{\kappa}[\ell] = \sigma'.\boldsymbol{\kappa}[\ell]$. As the rules in non-singleton looplets do not change shared variables, $\mathsf{crep}_c^{\Omega}[\sigma, \tau](\sigma).\mathbf{g} = \sigma.\mathbf{g} = \sigma'.\mathbf{g}$. Therefore, $\mathsf{crep}_c^{\Omega}[\sigma, \tau](\sigma) = \sigma'$. □

5.3 Representatives for one context and one looplet

We now summarize results from Sects. 5.1 and 5.2, giving the representative of a schedule τ in the case when τ uses only the rules from one looplet, and does not change its context. If the given looplet consists of a single rule, the construction is given in Sect. 5.1, and otherwise in Sect. 5.2. We show that these constructions indeed give us a schedule of bounded length, that reaches the same state as τ.

In the following, given a threshold automaton TA and a looplet c, we will say that a *schedule* $\tau = t_1, \ldots, t_n$ *is contained in* c, if $[t_i.rule] = c$ for $1 \le i \le n$.

Theorem 5.1 *Fix a threshold automaton, and a context Ω, and a looplet c in the slice $\mathsf{TA}|_{\Omega}$. Let σ be a configuration and τ be a steady schedule contained in c and applicable to σ. There exists a representative schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ with the following properties:*

(a) *schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ is applicable to σ, and $\mathsf{crep}_c^{\Omega}[\sigma, \tau](\sigma) = \tau(\sigma)$,*
(b) *the rule of each transition t in $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ belongs to c, that is, $[t.rule] = c$,*
(c) *schedule $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ is not longer than $2 \cdot |c|$.*

Proof If $|c| = 1$, then we use a single accelerated transition or the empty schedule as representative, as described in Lemma 5.1.

If $|c| > 1$, we construct the representative as in Sect. 5.2, so that by Lemma 5.8 property (a) follows. For every edge $e \in E_c$, the rule $L_c(e)$ belongs to c, and thus $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ satisfies property (b). As $|E_{\mathsf{in}}| \le |c|$ and $|E_{\mathsf{out}}| \le |c|$, we conclude that $|\mathsf{crep}_c^{\Omega}[\sigma, \tau]| \le 2 \cdot |c|$, and thus property c) is also satisfied. From this and Lemma 5.8, we conclude that $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ is the required representative schedule. □

Theorem 5.1 gives us a way to construct schemas that generate all representatives of the schedules contained in a looplet:

Theorem 5.2 *Fix a threshold automaton TA, a context Ω, and a looplet c in the slice $\mathsf{TA}|_{\Omega}$. There exists a schema $\mathsf{cschema}_c^{\Omega}$ with the following properties:*

Fix an arbitrary configuration σ and a steady schedule τ that is contained in c and is applicable to σ. Let $\tau' = \mathsf{crep}_c^{\Omega}[\sigma, \tau]$ be the representative schedule of τ, from Theorem 5.1. Then, $\mathsf{path}(\sigma, \tau')$ is generated by $\mathsf{cschema}_c^{\Omega}$. Moreover, the length of $\mathsf{cschema}_c^{\Omega}$ is at most $2 \cdot |c|$.

Proof Note that $\tau' = \mathsf{crep}_c^{\Omega}[\sigma, \tau]$ can be constructed in two different ways depending on the looplet c.

If $|c| = 1$, then by Lemma 5.1 we have that $\tau' = (r, f)$ for a rule $r \in c$ and a factor $f \in \mathbb{N}_0$. In this case we construct $\mathsf{cschema}_c^{\Omega}$ to be

$$\mathsf{cschema}_c^{\Omega} = \{\Omega\}\, r\, \{\Omega\}.$$

It is easy to see that $\mathsf{path}(\sigma, \tau')$ is generated by $\mathsf{cschema}_c^{\Omega}$, as well as that the length of $\mathsf{cschema}_c^{\Omega}$ is exactly 1, that is less than $2 \cdot |c|$.

Springer

If $|c| > 1$, then we use the trees T_{in} and T_{out} to construct the schema $\mathsf{cschema}_c^{\Omega}$ as follows:

$$\mathsf{cschema}_c^{\Omega} = \{\Omega\}\, r_{\mathsf{in}}(1) \cdots r_{\mathsf{in}}(|E_{\mathsf{in}}|) \cdot r_{\mathsf{out}}(1) \cdots r_{\mathsf{out}}(|E_{\mathsf{out}}|)\, \{\Omega\}. \tag{5.9}$$

Since for an arbitrary configuration σ and a schedule τ, we use the same sequence of edges in Eqs. (5.4) and (5.9) to construct $\mathsf{crep}_c^{\Omega}[\sigma, \tau]$ and $\mathsf{cschema}_c^{\Omega}$, the schema $\mathsf{cschema}_c^{\Omega}$ generates all paths of the representative schedules, and its length is at most $2 \cdot |c|$. □

6 Case II: one context and multiple looplets

In this section, we show that for each steady schedule, there exists a representative steady schedule of bounded length that reaches the same final state.

Theorem 6.1 *Fix a threshold automaton and a context Ω. For every configuration σ with $\omega(\sigma) = \Omega$ and every steady schedule τ applicable to σ, there exists a steady schedule $\mathsf{srep}_{\Omega}[\sigma, \tau]$ with the following properties:*

(a) $\mathsf{srep}_{\Omega}[\sigma, \tau]$ *is applicable to σ, and* $\mathsf{srep}_{\Omega}[\sigma, \tau](\sigma) = \tau(\sigma)$,
(b) $|\mathsf{srep}_{\Omega}[\sigma, \tau]| \leq 2 \cdot |(\mathcal{R}|_{\Omega})|$

To construct a representative schedule, we fix a context Ω of a TA, a configuration σ with $\omega(\sigma) = \Omega$, and a steady schedule τ applicable to σ. The key notion in our construction is a projection of a schedule on a set of looplets:

Definition 6.1 Let $\tau = t_1, \ldots, t_k$, for $k > 0$, be a schedule, and let C be a set of looplets. Given an increasing sequence of indices $i(1), \ldots, i(m) \in \{1, \ldots, k\}$, where $m \leq k$, i.e., $i(j) < i(j+1)$, for $1 \leq j < m$, a schedule $t_{i(1)} \ldots t_{i(m)}$ is a projection of τ on C, if each index $j \in \{1, \ldots, k\}$ belongs to $\{i(1), \ldots, i(m)\}$ if and only if $[t_j.\mathit{rule}] \in C$.

In fact, each schedule τ has a unique projection on a set C. In the following, we write $\tau|_{c_1,\ldots,c_m}$ to denote the projection of τ on a set $\{c_1, \ldots, c_m\}$.

Provided that $c_1, \ldots, c_m$ are all looplets of the slice $\mathcal{R}|_{\Omega}$ ordered with respect to $\prec_C^{lin}$, we construct the following sequences of projections on each looplet (note that π_0 is the empty schedule): $\pi_i = \tau|_{c_1} \cdot \cdots \cdot \tau|_{c_i}$ for $0 \leq i \leq m$.

Having defined $\{\pi_i\}_{0 \leq i \leq m}$, we construct the representative $\mathsf{srep}_{\Omega}[\sigma, \tau]$ simply as a concatenation of the representatives of each looplet:

$$\mathsf{srep}_{\Omega}[\sigma, \tau] = \mathsf{crep}_{c_1}^{\Omega}[\pi_0(\sigma), \tau|_{c_1}] \cdot \mathsf{crep}_{c_2}^{\Omega}[\pi_1(\sigma), \tau|_{c_2}] \cdot \ldots \cdot \mathsf{crep}_{c_m}^{\Omega}[\pi_{m-1}(\sigma), \tau|_{c_m}]$$

Example 6.1 Consider the TA shown in Fig. 8. It has three looplets, namely $c_1 = \{r_1, r_2, r_3, r_4\}$, $c_2 = \{r_5\}$, $c_3 = \{r_6, r_7, r_8\}$, and the rules are depicted as solid, dotted, and dashed, respectively. These looplets are ordered such that $c_1 \prec_C^{lin} c_2 \prec_C^{lin} c_3$.

Let σ be the configuration represented in Fig. 8 left, i.e. $\boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = \boldsymbol{\kappa}[\ell_5] = 1$ and $\boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = \boldsymbol{\kappa}[\ell_5] = 0$. Let τ be the schedule $(r_4, 1), (r_6, 1), (r_3, 1), (r_4, 1), (r_1, 1), (r_2, 1), (r_7, 1), (r_3, 1), (r_1, 1), (r_5, 1), (r_7, 1), (r_4, 1), (r_8, 1), (r_1, 1), (r_6, 1), (r_7, 1), (r_5, 1), (r_8, 1), (r_7, 1)$. Note that τ is applicable to σ and that $\tau(\sigma)$ is the configuration σ' from Fig. 8 right, i.e. $\boldsymbol{\kappa}[\ell_5] = 1$, $\boldsymbol{\kappa}[\ell_6] = 2$ and $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_2] = \boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = 0$. We construct the representative schedule $\mathsf{srep}_{\Omega}[\sigma, \tau]$.

Projection of τ on the looplets c_1, c_2, and c_3, gives us the following schedules:

$$\tau|_{c_1} = (r_4, 1), (r_3, 1), (r_4, 1), (r_1, 1), (r_2, 1), (r_3, 1), (r_1, 1), (r_4, 1), (r_1, 1),$$

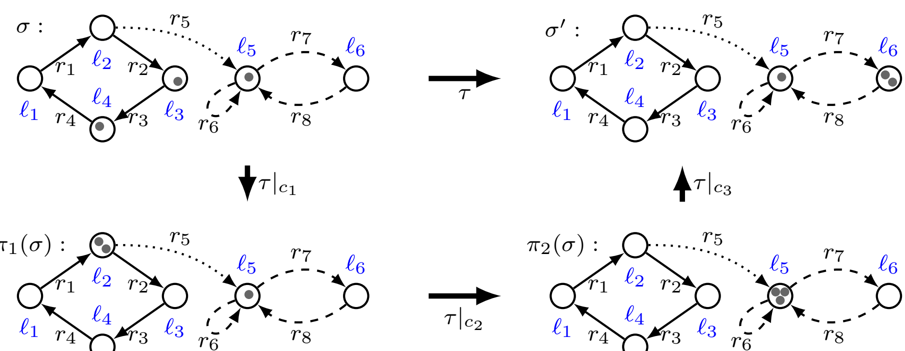


Fig. 8 Threshold automaton and configurations used in Example 6.1

$$\tau|_{c_2} = (r_5, 1), (r_5, 1),$$
$$\tau|_{c_3} = (r_6, 1), (r_7, 1), (r_7, 1), (r_8, 1), (r_6, 1), (r_7, 1), (r_8, 1), (r_7, 1).$$

Recall that

$$\mathsf{srep}_\Omega[\sigma, \tau] = \mathsf{crep}^\Omega_{c_1}[\pi_0(\sigma), \tau|_{c_1}] \cdot \mathsf{crep}^\Omega_{c_2}[\pi_1(\sigma), \tau|_{c_2}] \cdot \mathsf{crep}^\Omega_{c_3}[\pi_2(\sigma), \tau|_{c_3}].$$

In order to construct this schedule, we firstly construct the required configurations. Note that $\pi_0(\sigma) = \sigma$. Then $\pi_1(\sigma) = \tau|_{c_1}(\sigma)$, and this is the configuration from Fig. 8 lower left, i.e. $\boldsymbol{\kappa}[\ell_2] = 2$, $\boldsymbol{\kappa}[\ell_5] = 1$ and $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_3] = \boldsymbol{\kappa}[\ell_4] = \boldsymbol{\kappa}[\ell_6] = 0$. Configuration $\pi_2(\sigma) = \tau|_{c_1} \cdot \tau|_{c_2}(\sigma) = \tau|_{c_2}(\pi_1(\sigma))$ is represented on Fig. 8 lower right, i.e. $\boldsymbol{\kappa}[\ell_5] = 3$ and all other counters are zero.

Section 5 deals with the construction of representatives of schedules that contain rules from only one looplet. Recall that construction of $\mathsf{crep}^\Omega_{c_1}[\pi_0(\sigma), \tau|_{c_1}]$ corresponds to the one from Example 5.2. Thus, we know that

$$\mathsf{crep}^\Omega_{c_1}[\pi_0(\sigma), \tau|_{c_1}] = (r_2, 0), (r_3, 1), (r_4, 2), (r_1, 2), (r_2, 0), (r_3, 0).$$

As c_2 is a singleton looplet, we use the result of Sect. 5.1. Thus,

$$\mathsf{crep}^\Omega_{c_2}[\pi_1(\sigma), \tau|_{c_2}] = (r_5, 2).$$

Using the result from Sect. 5.2 we obtain that

$$\mathsf{crep}^\Omega_{c_3}[\pi_2(\sigma), \tau|_{c_3}] = (r_8, 0), (r_7, 2),$$

and finaly we have the representative for τ that is

$$\mathsf{srep}_\Omega[\sigma, \tau] = (r_2, 0), (r_3, 1), (r_4, 2), (r_1, 2), (r_2, 0), (r_3, 0), (r_5, 2), (r_8, 0), (r_7, 2).$$

Lemma 6.1 (Looplet sorting) *Given a threshold automaton, a context Ω, a configuration σ, a steady schedule τ applicable to σ, and a sequence $c_1, \ldots, c_m$ of all looplets in the slice $\mathcal{R}|_\Omega$ with the property $c_i \prec^{lin}_C c_j$ for $1 \le i < j \le m$, the following holds:*

1. *Schedule $\tau|_{c_1}$ is applicable to the configuration σ.*
2. *Schedule $\tau|_{c_2,\ldots,c_m}$ is applicable to the configuration $\tau|_{c_1}(\sigma)$.*
3. *Schedule $\tau|_{c_1} \cdot \tau|_{c_2,\ldots,c_m}$, when applied to σ, results in configuration $\tau(\sigma)$.*

Proof In the following, we show Points 1–3 one-by-one.

We need extra notation. For a local state ℓ we denote by $\mathbf{1}_\ell$ the $|\mathcal{L}|$-dimensional vector, where the ℓth component is 1, and all the other components are 0. Given a schedule $\rho = t_1 \cdots t_k$, we introduce a vector $\Delta_{\boldsymbol{\kappa}}(\rho) \in \mathbb{Z}^{|\mathcal{L}|}$ to keep counter difference and a vector $\Delta_{\mathbf{g}}(\rho) \in \mathbb{N}_0^{|\Gamma|}$ to keep difference on shared variables as follows:

$$\Delta_{\boldsymbol{\kappa}}(\rho) = \sum_{1 \le i \le |\rho|} t_i.\mathit{factor} \cdot (\mathbf{1}_{t_i.to} - \mathbf{1}_{t_i.\mathit{from}}) \quad \text{and} \quad \Delta_{\mathbf{g}}(\rho) = \sum_{1 \le i \le |\rho|} t_i.\mathbf{u}$$

Proof of (1) Assume by contradiction that schedule $\tau|_{c_1}$ is not applicable to configuration σ. Thus, there is a schedule τ' and a transition t^* that constitute a prefix of $\tau|_{c_1}$, with the following property: τ' is applicable to σ, whereas $\tau' \cdot t^*$ is not applicable to σ. Let $\ell = t^*.\mathit{from}$ and $\ell' = t^*.to$.

There are three cases of why t^* may be not applicable to $\tau'(\sigma)$:

(i) There is not enough processes to move: $(\sigma.\boldsymbol{\kappa} + \Delta_{\boldsymbol{\kappa}}(\tau' \cdot t^*))[\ell] < 0$. As τ is applicable to σ, there is a transition t of τ with $[t.\mathit{rule}] \ne c_1$ and $t.to = \ell$ as well as $t.\mathit{factor} > 0$. From this, by definition of $\prec_C^{lin}$, it follows that $[t.\mathit{rule}] \prec_C^{lin} c_1$. This contradicts the lemma's assumption on the order $c_1 \prec_C^{lin} \cdots \prec_C^{lin} c_m$.

(ii) The condition $t^*.\varphi^{\text{rise}}$ is not satisfied, that is, $\tau'(\sigma) \not\models t^*.\varphi^{\text{rise}}$. Then, there is a guard $\varphi \in \mathsf{guard}(t^*.\varphi^{\text{rise}})$ with $\tau'(\sigma) \not\models \varphi$.

Since τ is applicable to σ, there is a prefix $\rho \cdot t$ of τ, for a schedule ρ and a transition t that unlocks φ in $\rho(\sigma)$, that is, $\rho(\sigma) \not\models \varphi$ and $t(\rho(\sigma)) \models \varphi$. Thus, transition t changes the context: $\omega(\rho(\sigma)) \ne \omega(t(\rho(\sigma)))$. This contradicts the assumption that schedule τ is steady.

(iii) The condition $t^*.\varphi^{\text{fall}}$ is not satisfied: $\tau'(\sigma) \not\models t^*.\varphi^{\text{fall}}$. Then, there is a guard $\varphi \in \mathsf{guard}(t^*.\varphi^{\text{fall}})$ with $\tau'(\sigma) \not\models \varphi$.

Let ρ be the longest prefix of τ satisfying $\rho|_{c_1} = \tau'$. Note that $\rho \cdot t^*$ is also a prefix of τ. As $\rho|_{c_1} = \tau'$ and no transition decrements the shared variables, we conclude that $(\tau'(\sigma)).\mathbf{g} \le (\rho(\sigma)).\mathbf{g}$. From this and from the fact that $\tau'(\sigma) \not\models \varphi$, it follows that $\rho(\sigma) \not\models \varphi$. Thus transition t^* is not applicable to $\rho(\sigma)$. This contradicts the assumption that τ is applicable to σ.

From (i), (ii), and (iii), we conclude that *(1)* holds.

Proof of (2) We show that $\tau|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$.

To this end, we fix an arbitrary prefix τ' of τ, a transition t, and a suffix τ'', that constitute τ, that is, $\tau = \tau' \cdot t \cdot \tau''$. We show that if schedule $\tau'|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$, then so is $(\tau' \cdot t)|_{c_2,\dots,c_m}$.

Let us assume that $\tau'|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$, and let σ'' denote the resulting state $(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma)$. We consider two cases:

- $[t.\mathit{rule}] = c_1$. This case holds trivially, as $(\tau' \cdot t)|_{c_2,\dots,c_m}$ equals to $\tau'|_{c_2,\dots,c_m}$, which is applicable to $\tau|_{c_1}(\sigma)$ by assumption.
- $[t.\mathit{rule}] \ne c_1$. In order to prove that$(\tau' \cdot t)|_{c_2,\dots,c_m}$ is applicable to $\tau|_{c_1}(\sigma)$, we show that counters $\sigma''.\boldsymbol{\kappa}$ and shared variables $\sigma''.\mathbf{g}$ are large enough, so that transition t is applicable to σ'':

(i) We start by showing that $\sigma''.\boldsymbol{\kappa}[t.\mathit{from}] \ge t.\mathit{factor}$. We distinguish between different cases on source and target states of transition t.

(i.A) We will show by contradiction that there is no rule $r \in c_1$ with $t.to = r.\mathit{from}$. Let's assume it exists. Then, on one hand, as $[t.\mathit{rule}] \ne c_1$, by definition of $\prec_C^{lin}$, it follows that $[t.\mathit{rule}] \prec_C^{lin} \dots \prec_C^{lin} c_1$. On the other hand, as $[t.\mathit{rule}] \ne c_1$ and $c_1, \dots, c_m$ are all classes of the rules used in τ, it holds that $[t.\mathit{rule}] \in \{c_2, \dots, c_m\}$. By the lemma's

assumption, $c_1 \prec_C^{lin} \cdots \prec_C^{lin} c_m$, and thus, $c_1 \prec_C^{lin} \cdots \prec_C^{lin} [t.rule]$. We arrive at a contradiction.

(i.B) Let's consider the case of a rule $r \in c_1$ with $r.to = t.from$. Assume by contradiction that t is not applicable to σ'', that is, $\sigma''.\boldsymbol{\kappa}[t.from] < t.factor$. On one hand, transition t is not applicable to $\sigma'' = (\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma)$. Then by the definition of $\Delta_{\boldsymbol{\kappa}}$, it holds that $\sigma[t.from] + (\Delta_{\boldsymbol{\kappa}}(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}) + \Delta_{\boldsymbol{\kappa}}(t))[t.from] < 0$. By observing that $\tau|_{c_1} = \tau'|_{c_1} + \tau''|_{c_1}$, we derive the following inequality:

$$\sigma[t.from] \\ +(\Delta_{\boldsymbol{\kappa}}(\tau'|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau''|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau'|_{c_2,\dots,c_m}) + \Delta_{\boldsymbol{\kappa}}(t))[t.from] < 0 \quad (6.1)$$

On the other hand, schedule $\tau = \tau' \cdot t \cdot \tau''$ is applicable to configuration σ. Thus, $\sigma[t.from] + (\Delta_{\boldsymbol{\kappa}}(\tau') + \Delta_{\boldsymbol{\kappa}}(t) + \Delta_{\boldsymbol{\kappa}}(\tau''))[t.from] \geq 0$. By observing that $\tau|_{c_1} = \tau'|_{c_1} + \tau''|_{c_1}$ and $\tau|_{c_2,\dots,c_m} = \tau'|_{c_2,\dots,c_m} + \tau''|_{c_2,\dots,c_m}$, we arrive at:

$$\sigma[t.from] + (\Delta_{\boldsymbol{\kappa}}(\tau'|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau'|_{c_2,\dots,c_m}) \\ +\Delta_{\boldsymbol{\kappa}}(t) + \Delta_{\boldsymbol{\kappa}}(\tau''|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau''|_{c_2,\dots,c_m}))[t.from] \geq 0 \quad (6.2)$$

By subtracting (6.2) from (6.1), and by commutativity of vector addition, we arrive at $\Delta_{\boldsymbol{\kappa}}(\tau''|_{c_2,\dots,c_m})[t.from] > 0$. Thus, there is a transition t' in $\tau''|_{c_2,\dots,c_m}$ and a rule $r' \in c_1$ such that $t'.to = r'.from$. We again arrived at the contradictory Case (i.A). Hence, transition t must be applicable to configuration σ''.

(i.C) Otherwise, neither $t.from$ nor $t.to$ belong to the set of local states affected by the rules from c_1, i.e., $\{t.from, t.to\} \cap \{\ell \mid \exists r \in c_1.\ r.from = \ell \lor r.to = \ell\}$ is empty. Then, schedule $\tau|_{c_1}$ does not change the counter $\boldsymbol{\kappa}[t.from]$, and $\Delta_{\boldsymbol{\kappa}}(\tau')[t.from] = \Delta_{\boldsymbol{\kappa}}(\tau'|_{c_2,\dots,c_m})[t.from]$. As t is applicable to $\tau'(\sigma)$, that is, $(\tau'(\sigma)).\boldsymbol{\kappa}[t.from] \geq t.factor$, we conclude that $\sigma''.\boldsymbol{\kappa}[t.from] \geq t.factor$.

(ii) We now show that $\sigma'' \models t.\varphi^{\text{rise}} \land t.\varphi^{\text{fall}}$. Assume by contradiction that $\sigma'' \not\models t.\varphi^{\text{rise}} \land t.\varphi^{\text{fall}}$. There are two cases to consider.

If $\sigma'' \not\models t.\varphi^{\text{rise}}$. By definition, the shared variables are never decremented in a non-singleton looplet. As τ' is a prefix of τ, schedule $\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}$ includes all transitions of τ'. Thus, $\Delta_{\mathbf{g}}(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}) \geq \Delta_{\mathbf{g}}(\tau')$. From this and $\sigma'' \not\models t.\varphi^{\text{rise}}$, it follows that $\tau'(\sigma) \not\models t.\varphi^{\text{rise}}$. This contradicts applicability of τ to σ.

If $\sigma'' \not\models t.\varphi^{\text{fall}}$. Then, there is a guard $\varphi \in \mathsf{guard}(t.\varphi^{\text{fall}})$ with $\tau''(\sigma) \not\models \varphi$. On one hand, $\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m}$ is applicable to σ. On the other hand, τ is applicable to σ. We notice that $\Delta_{\mathbf{g}}(\tau) = \Delta_{\mathbf{g}}(\tau|_{c_1}) + \Delta_{\mathbf{g}}(\tau'|_{c_2,\dots,c_m}) + \Delta_{\mathbf{g}}(\tau''|_{c_2,\dots,c_m}) + \Delta_{\mathbf{g}}(t) \geq \Delta_{\mathbf{g}}(\tau|_{c_1}) + \Delta_{\mathbf{g}}(\tau'|_{c_2,\dots,c_m})$. As shared variables are never decreased, it follows that $(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma) \not\models \varphi$. Thus, $\omega(\sigma) \neq \omega(\tau(\sigma))$. This contradicts the assumption on that schedule τ is steady.

Having proved that, we conclude that transition t is applicable to configuration $(\tau|_{c_1} \cdot \tau'|_{c_2,\dots,c_m})(\sigma)$. Thus, by induction $(\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m})(\sigma)$ is applicable to σ. We conclude that Point *2* of the theorem holds.

Proof of (3) By the commutativity property of vector addition,

$$\Delta_{\boldsymbol{\kappa}}(\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m}) = \Delta_{\boldsymbol{\kappa}}(\tau|_{c_1}) + \Delta_{\boldsymbol{\kappa}}(\tau|_{c_2,\dots,c_m}) = \sum_{1 \leq i \leq |\tau|} \Delta_{\boldsymbol{\kappa}}(t_i) = \Delta_{\boldsymbol{\kappa}}(\tau).$$

Thus, $(\tau|_{c_1} \cdot \tau|_{c_2,\dots,c_m})(\sigma) = \tau(\sigma)$, and Point (3) follows.

We have thus shown all three points of Lemma 6.1. □

Proof (*of Theorem* 6.1) By iteratively applying Lemma 6.1, we prove by induction that schedule $\tau|_{c_1} \cdot \ldots \cdot \tau|_{c_m}$ is applicable to σ and results in $\tau(\sigma)$. From Theorem 5.1, we conclude that each schedule $\tau|_{c_i}$ can be replaced by its representative $\mathsf{crep}^{\Omega}_{c_i}[\pi_{i-1}(\sigma), \tau|_{c_i}]$. Thus, $\mathsf{srep}_{\Omega}[\sigma, \tau]$ is applicable to σ and results in $\tau(\sigma)$. By Proposition 3.4, schedule $\mathsf{srep}_{\Omega}[\sigma, \tau]$ is steady, since $\omega(\sigma) = \omega(\tau(\sigma))$. □

Finally, we show that for a given context, there is a schema that generates all paths of such representative schedules.

Theorem 6.2 *Fix a threshold automaton and a context Ω. Let $c_1, \ldots, c_m$ be the sorted sequence of all looplets of the slice $\mathcal{R}|_{\Omega}$, i.e., $c_1 \prec^{lin}_C \ldots \prec^{lin}_C c_m$. Schema $\mathsf{sschema}_{\Omega} = \mathsf{cschema}^{\Omega}_{c_1} \circ \cdots \circ \mathsf{cschema}^{\Omega}_{c_m}$ has two properties: (a) For a configuration σ with $\omega(\sigma) = \Omega$ and a steady schedule τ applicable to σ, $\mathsf{path}(\sigma, \tau')$ of the representative $\tau' = \mathsf{srep}_{\Omega}[\sigma, \tau]$ is generated by $\mathsf{sschema}_{\Omega}$; and (b) the length of $\mathsf{sschema}_{\Omega}$ is at most $2 \cdot |(\mathcal{R}|_{\Omega})|$.*

Proof Fix a configuration σ with $\omega(\sigma) = \Omega$ and a steady schedule τ applicable to σ. As $\mathsf{srep}_{\Omega}[\sigma, \tau]$ is a sorted sequence of the looplet representatives, all paths of $\mathsf{srep}_{\Omega}[\sigma, \tau]$ are generated by $\mathsf{sschema}_{\Omega}$, which is not longer than $2 \cdot |(\mathcal{R}|_{\Omega})|$. □

7 Proving the main result

Using the results from Sects. 5 and 6, for each configuration and each schedule (without restrictions) we construct a representative schedule.

Theorem 7.1 *Given a threshold automaton, a configuration σ, and a schedule τ applicable to σ, there exists a schedule $\mathsf{rep}[\sigma, \tau]$ with the following properties:*

(a) $\mathsf{rep}[\sigma, \tau]$ *is applicable to* σ, *and* $\mathsf{rep}[\sigma, \tau](\sigma) = \tau(\sigma)$,
(b) $|\mathsf{rep}[\sigma, \tau]| \leq 2 \cdot |\mathcal{R}| \cdot (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}| + 1) + |\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|$.

Proof Given a threshold automaton, fix a configuration σ and a schedule τ applicable to σ. Let $\Omega_1, \ldots, \Omega_{K+1}$ be the maximal monotonically increasing sequence of contexts such that $\mathsf{path}(\sigma, \tau)$ is consistent with the sequence by Definition 3.7. From Proposition 3.2, the length of the sequence is $K + 1 = |\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}| + 1$. Thus, there are at most K transitions $t^{\star}_1, \ldots, t^{\star}_K$ in τ that change their context, i.e., for $i \in \{1, \ldots, K\}$, it holds $\omega(\sigma_i) \sqsubset \omega(t^{\star}_i(\sigma_i))$ for $t^{\star}_i$'s respective state σ_i in τ. Therefore, we can divide τ into $K + 1$ steady schedules separated by the transitions $t^{\star}_1, \ldots, t^{\star}_K$:

$$\tau = \nu_1 \cdot t^{\star}_1 \cdot \nu_2 \cdots \nu_K \cdot t^{\star}_K \cdot \nu_{K+1}.$$

Now, the main idea is to replace the steady schedules with their representatives from Theorem 6.1. That is, using $t^{\star}_1, \ldots, t^{\star}_K$ and $\nu_1, \ldots, \nu_{K+1}$, we construct the schedules $\rho_1, \ldots, \rho_K$ (by convention, ρ_0 is the empty schedule):

$$\rho_i = \rho_{i-1} \cdot \nu_i \cdot t^{\star}_i \quad \text{for } 1 \leq i \leq K.$$

Finally, the representative schedule $\mathsf{rep}[\tau, \sigma]$ is constructed as follows:

$$\mathsf{rep}_{\Omega_1}[\sigma, \nu_1] \cdot t^{\star}_1 \cdot \mathsf{rep}_{\Omega_2}[\rho_1(\sigma), \nu_2] \cdots \mathsf{rep}_{\Omega_K}[\rho_{K-1}(\sigma), \nu_K] \cdot t^{\star}_K \cdot \mathsf{rep}_{\Omega_{K+1}}[\rho_K(\sigma), \nu_{K+1}]$$

From Theorem 6.1, it follows that $\mathsf{rep}[\tau, \sigma]$ is applicable to σ and it results in $\tau(\sigma)$. Moreover, the representative of a steady schedule is not longer than $2|\mathcal{R}|$, which together with K transitions gives us the bound $2|\mathcal{R}|(K+1)+K$. As we have that $K = |\Phi^{\mathrm{rise}}|+|\Phi^{\mathrm{fall}}|$, this gives us the required bound. □

Further, given a maximal monotonically increasing sequence z of contexts, we construct a schema that generates all paths of the schedules consistent with z:

Theorem 7.2 *For a threshold automaton and a monotonically increasing sequence z of contexts, there exists a schema* $\mathsf{schema}(z)$ *that generates all paths of the representative schedules that are consistent with z, and the length of* $\mathsf{schema}(z)$ *does not exceed* $3 \cdot |\mathcal{R}| \cdot (|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|) + 2 \cdot |\mathcal{R}|$.

Proof Given a threshold automaton, let ρ_{all} be the sequence $r_1, \ldots, r_{|\mathcal{R}|}$ of all rules from $\mathcal{R}$, and let $z = \Omega_0, \ldots, \Omega_m$ be a monotonically increasing sequence of contexts. By the construction in Theorem 7.1, each representative schedule $\mathsf{rep}[\sigma, \tau]$ consists of the representatives of steady schedules terminated with transitions that change the context. Then, for each context Ω_i, for $0 \le i < m$, we compose $\mathsf{sschema}_{\Omega}$ and $\{\Omega_i\}\, \rho_{\mathsf{all}}\, \{\Omega_{i+1}\}$. This composition generates the representative of a steady schedule and the transition changing the context from Ω_i to Ω_{i+1}. Consequently, we construct the $\mathsf{schema}(z)$ as follows:

$$(\mathsf{sschema}_{\Omega_0} \circ \{\Omega_0\}\, \rho_{\mathsf{all}}\, \{\Omega_1\}) \circ \ldots \circ (\mathsf{sschema}_{\Omega_{m-1}} \circ \{\Omega_{m-1}\}\, \rho_{\mathsf{all}}\, \{\Omega_m\}) \circ \mathsf{sschema}_{\Omega_m}$$

By inductively applying Theorem 6.2, we prove that $\mathsf{schema}(z)$ generates all paths of schedules $\mathsf{rep}[\sigma, \tau]$ that are consistent with the sequence z. We get the needed bound on the length of $\mathsf{schema}(z)$ by using an argument similar to Theorem 7.1 and by noting that for every context, instead of one rule that is changing it, we add $|\mathcal{R}|$ extra rules. □

8 Complete set of schemas and optimizations

Our proofs show that the set of schemas is easily computed from the TA: the threshold guards are syntactic parts of the TA, and enable us to directly construct increasing sequences of contexts. To find a slice of the TA for a given context, we filter the rules with unlocked guards, i.e., check whether the context contains the guard. To produce the simple schema of a looplet, we compute a spanning tree over the slice. To construct simple schemas, we do a topological sort over the looplets. For example, it takes just 30 s to compute the schemas in our longest experiment that runs for 4 h. In our tool we have implemented the following optimizations that lead to simpler and fewer SMT queries.

Entailment optimization We say that a guard $\varphi_1 \in \Phi^{\mathrm{rise}}$ *entails* a guard $\varphi_2 \in \Phi^{\mathrm{rise}}$, if for all combinations of parameters $\mathbf{p} \in \mathbf{P}_{RC}$ and shared variables $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$, it holds that $(\mathbf{g}, \mathbf{p}) \models \varphi_1 \rightarrow \varphi_2$. For instance, in our example, $\varphi_3 \colon y \ge (2t+1) - f$ entails $\varphi_2 \colon y \ge (t+1) - f$. If φ_1 entails φ_2, then we can omit all monotonically increasing sequences that contain a context $(\Omega^{\mathrm{rise}}, \Omega^{\mathrm{fall}})$ with $\varphi_1 \in \Omega^{\mathrm{rise}}$ and $\varphi_2 \notin \Omega^{\mathrm{rise}}$. If the number of schemas before applying this optimization is $m!$ and there are k entailments, then the number of schemas reduces from $m!$ to $(m-k)!$. A similar optimization is introduced for the guards from Φ^{fall}.

Control flow optimization Based on the proof of Lemma 6.1, we introduce the following optimization for TAs that are directed acyclic graphs (possibly with self loops). We say that a rule $r \in \mathcal{R}$ *may unlock* a guard $\varphi \in \Phi^{\mathrm{rise}}$, if there is a $\mathbf{p} \in \mathbf{P}_{RC}$ and $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ satisfying: $(\mathbf{g}, \mathbf{p}) \models r.\varphi^{\mathrm{rise}} \wedge r.\varphi^{\mathrm{fall}}$ (the rule is unlocked); $(\mathbf{g}, \mathbf{p}) \not\models \varphi$ (the guard is locked); $(\mathbf{g} + r.\mathbf{u}, \mathbf{p}) \models \varphi$ (the guard is now unlocked).

In our example from Fig. 2, the rule $r_1 \colon \mathit{true} \mapsto x{+}{+}$ may unlock the guard $\varphi_1 \colon x \ge \lceil (n+t)/2 \rceil - f$.

Let $\varphi \in \Phi^{\mathrm{rise}}$ be a guard, $r'_1, \dots, r'_m$ be the rules that use φ, and $r_1, \dots, r_k$ be the rules that may unlock φ. If $r_i \prec_C^{lin} r'_j$, for $1 \le i \le k$ and $1 \le j \le m$, then we exclude some sequences of contexts as follows (we call φ *forward-unlockable*). Let $\psi_1, \dots, \psi_n \in \Phi^{\mathrm{rise}}$ be the guards of $r_1, \dots, r_k$. Guard φ cannot be unlocked before $\psi_1, \dots, \psi_n$, and thus we can omit all sequences of contexts, where φ appears in the contexts before $\psi_1, \dots, \psi_n$. Moreover, as $\psi_1, \dots, \psi_n$ are the only guards of the rules unlocking φ, we omit the sequences with different combinations of contexts involving φ and the guards from $\Phi^{\mathrm{rise}} \setminus \{\varphi, \psi_1, \dots, \psi_n\}$. Finally, as the rules $r'_1, \dots, r'_m$ appear after the rules $r_1, \dots, r_k$ in the order $\prec_C^{lin}$, the rules $r'_1, \dots, r'_m$ appear after the rules $r_1, \dots, r_k$ in a rule sequence of every schema. Thus, we omit the combinations of the contexts involving φ and $\psi_1, \dots, \psi_n$.

Hence, we add all forward-unlockable guards to the initial context (we still check the guards of the rules in the SMT encoding in Sect. 9). If the number of schemas before applying this optimization is $m!$ and there are k forward-unlocking guards, then the number of schemas reduces from $m!$ to $(m-k)!$. A similar optimization is introduced for the guards from Φ^{fall}.

9 Checking a schema with SMT

We decompose a schema into a sequence of simple schemas, and encode the simple schemas. Given a simple schema $S = \{\Omega_1\}\, r_1, \dots, r_m\, \{\Omega_2\}$, which contains m rules, we construct an SMT formula such that every model of the formula represents a path from $\mathcal{L}(S)$—the language of paths generated by schema S—and for every path in $\mathcal{L}(S)$ there is a corresponding model of the formula. Thus, we need to model a path of $m+1$ configurations and m transitions (whose acceleration factors may be 0).

To represent a configuration σ_i, for $0 \le i \le m$, we introduce two vectors of SMT variables: Given the set of local states $\mathcal{L}$ and the set of shared variables Γ, a vector $\mathbf{k}^i = (k^i_1, \dots, k^i_{|\mathcal{L}|})$ to represent the process counters, a vector $\mathbf{x}^i = (x^i_1, \dots, x^i_{|\Gamma|})$ to represent the shared variables. We call the pair $(\mathbf{k}^i, \mathbf{x}^i)$ the *layer* i, for $1 \le i \le m$.

Based on this we encode schemas, for which the sequence of rules $r_1, \dots, r_m$ is fixed. We exploit this in two ways: First, we encode for each layer i the constraints of rule r_i. Second, as this constraint may update only two counters—the processes move from and move to according to the rule—we do not need $|\mathcal{L}|$ counter variables per layer, but only encode the two counters per layer that have actually changed. As is a common technique in bounded model checking, the counters that are not changed are "reused" from previous layers in our encoding. By doing so, we encode the schema rules with $|\mathcal{L}| + |\Gamma| + m \cdot (2 + |\Gamma|)$ integer variables, $2m$ equations, and inequalities in linear integer arithmetic that represent threshold guards that evaluate to true (at most the number of threshold guards times m of these inequalities).

In the following, we use the notation $[k : m]$ to denote the set $\{k, \dots, m\}$. In order to reuse the variables from the previous layers, we introduce a function $\upsilon : \mathcal{L} \times [0 : m] \to [0 : m]$ that for a layer $i \in [0 : m]$ and a local state $\ell \in \mathcal{L}$, gives the largest number $j \le i$ of the layer, where the counter k^j_ℓ is updated:

$$\upsilon(\ell, i) = \begin{cases} i, & \text{if } i = 0 \lor \ell \in \{r_i.\mathit{from}, r_i.\mathit{to}\} \\ \upsilon(\ell, i-1), & \text{otherwise.} \end{cases}$$

Having defined layers, we encode: the effect of rules on counters and shared variables (in formulas M and U below), the effect of rules on the configuration (T), restrictions imposed by contexts (C), and, finally, the reachability question.

To represent m transitions, for each transition $i \in [1 : m]$, we introduce a non-negative variable δ^i for the acceleration factor, and define two formulas: formula $M^\ell(i-1, i)$ to express the update of the counter of local state $\ell \in \mathcal{L}$, and formula $U^x(i-1, i)$ to represent the update of the shared variable $x \in \Gamma$:

$$M^\ell(i-1, i) \equiv \begin{cases} k_\ell^i = k_\ell^{\upsilon(\ell, i-1)} + \delta^i, & \text{for } \ell = r_i.\mathit{to} \text{ and } i \in [1 : m] \\ k_\ell^i = k_\ell^{\upsilon(\ell, i-1)} - \delta^i, & \text{for } \ell = r_i.\mathit{from} \text{ and } i \in [1 : m] \\ \mathit{true}, & \text{otherwise} \end{cases}$$

$$U^x(i-1, i) \equiv \begin{cases} x^i = x^{i-1} + \delta^i \cdot u, & \text{if } u = r_i.\mathbf{u}[j] > 0, \\ \mathit{true}, & \text{otherwise.} \end{cases}$$

The formula $T(i-1, i)$ collects all constraints by the rule r_i:

$$T(i-1, i) \equiv \bigwedge_{\ell \in \mathcal{L}} M^\ell(i-1, i) \land \bigwedge_{x \in \Gamma} U^x(i-1, i).$$

For a formula φ, we denote by $\varphi[\mathbf{x}^i]$ the formula, where each variable $x \in \Gamma$ is substituted with x^i. Then, given a context $\Omega = (\Omega^{\text{rise}}, \Omega^{\text{fall}})$, a formula $C^\Omega(i)$ adds the constraints of the context Ω on the layer i:

$$C_\Omega(i) \equiv \bigwedge_{\varphi \in \Omega^{\text{rise}}} \varphi[\mathbf{x}^i] \land \bigwedge_{\varphi \in \Phi^{\text{rise}} \setminus \Omega^{\text{rise}}} \neg\varphi[\mathbf{x}^i] \land \bigwedge_{\varphi \in \Omega^{\text{fall}}} \neg\varphi[\mathbf{x}^i] \land \bigwedge_{\varphi \in \Phi^{\text{fall}} \setminus \Omega^{\text{fall}}} \varphi[\mathbf{x}^i].$$

Finally, the formula $C_{\Omega_1}(0) \land T(0, 1) \land \cdots \land T(m-1, m) \land C_{\Omega_2}(m)$ captures all the constraints of the schema $S = \{\Omega_1\}\, r_1, \ldots, r_m\, \{\Omega_2\}$, and thus, its models correspond to the paths of schedules that are generated by S.

Let $I(0)$ be the formula over the variables of layer i that captures the initial states of the threshold automaton, and $B(i)$ be a state property over the variables of layer i. Then, parameterized reachability for the schema S is encoded with the following formula in linear integer arithmetic:

$$I(0) \land C_{\Omega_1}(0) \land T(0, 1) \land \cdots \land T(m-1, m) \land C_{\Omega_2}(m) \land \big(B(0) \lor \cdots \lor B(m)\big).$$

10 Experiments

We have extended our tool ByMC (Byzantine Model Checker [2]) with the technique discussed in this paper. All of our benchmark algorithms were originally published in pseudo-code, and we model them in a parametric extension of PROMELA, which was discussed in [27,34].

10.1 Benchmarks

We revisited several asynchronous FTDAs that were evaluated in [33,41]. In addition to these classic FTDAs, we considered asynchronous (Byzantine) consensus algorithms, namely,

BOSCO [57], C1CS [10], and CF1S [18], that are designed to work despite partial failure of the distributed system. In contrast to the conference version of this paper [39], we used a new version of the benchmarks from [37] that have been slightly updated for liveness properties. Hence, for some benchmarks, the running times of our tool may vary from [39]. The benchmarks, their source code in parametric of PROMELA, and the code of the threshold automata are freely available [30].

10.2 Implementation

ByMC supports several tool chains (shown in Fig. 1, p. 3), the first using counter abstraction (that is, process counters over an abstract domain), and the second using counter systems with counters over integers:

Data and counter abstractions In this chain, the message counters are first mapped to parametric intervals, e.g., counters range over the abstract domain $\hat{D} = \{[0, 1), [1, t + 1), [t + 1, n - t), [n - t, \infty)\}$. By doing so, we obtain a finite (data) abstraction of each process, and thus we can represent the system as a counter system: We maintain one counter $\kappa[\ell]$ per local state ℓ of a process, as well as the counters for the sent messages. Then, in the counter abstraction step, every process counter $\kappa[\ell]$ is mapped to the set of parametric intervals $\hat{D}$. As the abstractions may produce spurious counterexamples, we run them in an abstraction-refinement loop that incrementally prunes spurious transitions and unfair executions. More details on the data and counter abstractions and refinement can be found in [33]. In our experiments, we use two kinds of model checkers as backend:

1. *BDD* The counter abstraction is checked with nuXmv [11] using Binary Decision Diagrams (BDDs). For safety properties, the tool executes the command `check_invar`. In our experiments, we used the timeout of 3 days, as there was at least one benchmark that needed a bit more than a day to complete.
2. *BMC* The counter abstraction is checked with nuXmv using bounded model checking [6]. To ensure completeness (at the level of counter abstraction), we explore the computations of the length up to the diameter bounds that were obtained in [41]. To efficiently eliminate shallow spurious counterexamples, we first run the bounded model checker in the incremental mode up to length of 30. This is done by issuing the `nuXmv` command `check_ltlspec_sbmc_inc`, which uses the built-in SAT solver `MiniSAT`. Then, we run a single-shot SAT problem by issuing the `nuXmv` command `gen_ltlspec_sbmc` and checking the generated formula with the SAT solver `lingeling` [5]. In our experiments, we set the timeout to 1 day.

Reachability for threshold automata In this tool chain, to obtain a threshold automaton, our tool first applies data abstraction over the domain $\hat{D}$ to the PROMELA code, which abstracts the message counters that keep the number of messages received by every process, while the message counters for the sent messages are kept as integers. More details can be found in [40]. Having constructed a threshold automaton, we compare two verification approaches:

1. *PARA² Bounded model checking with SMT* The approach of this article. BYMC enumerates the schemas (as explained in Sect. 4), encodes them in SMT (as explained in Sect. 9) and checks every schema with the SMT solver `Z3` [17].

2. *FAST Acceleration of counter automata* In this chain, our tool constructs a threshold automaton and checks the reachability properties with the existing tool FAST [3]. For comparison with our tool, we run FAST with the MONA plugin that produced the best results in our experiments.

The challenge in the verification of FTDAs is the immense non-determinism caused by interleavings, asynchronous message passing, and faults. In our modeling, all these are reflected in non-deterministic choices in the PROMELA code. To obtain threshold automata, as required for our technique, our tool constructs a parametric interval data abstraction [33] that adds to non-determinism.

Comparing to [39], in this paper, we have introduced an optimization to schema checking that dramatically reduced the running times for some of the benchmarks. In this optimization, we group schemas in a prefix tree, whose nodes are contexts and edges are simple schemas. In each node of the prefix tree, our tool checks, whether there are configurations that are reachable from the initial configurations by following the schemas in the prefix. If there are no such reachable configurations, we can safely prune the whole suffix and thus prove many schemas to be unsatisfiable at once.

10.3 Evaluation

Table 1 summarizes the features of threshold automata that are automatically constructed by ByMC from parametric PROMELA. The number of local states $|\mathcal{L}|$ varies from 7 (FRB and STRB) to hundreds (C1CS and CBC). Our threshold automata are obtained by applying interval abstraction to PROMELA code, which keeps track of the number of messages received by each process. Thus, the number $|\mathcal{L}|$ is proportional to the number of control states and $|\widehat{D}|^k$, where $\widehat{D}$ is the domain of parametric intervals (discussed above) and k is the number of message types. Sometimes, one can manually construct a more efficient threshold automaton that models the same fault-tolerant distributed algorithm and preserves the same safety properties. For instance, Fig. 2 shows a manual abstraction of ABA that has only 5 local states, in contrast to 61 local states in the automatic abstraction (cf. Table 1). We leave open the question of whether one can automatically construct a minimal threshold automaton with respect to given specifications.

Table 2 summarizes our experiments conducted with the techniques introduced in Sect. 10.2: BDD, BMC, PARA^2, and FAST. On large problems, our new technique works significantly better than BDD- and SAT-based model checking. BDD-based model checking works very well on top of counter abstraction. Importantly, our new technique does not use abstraction refinement. In comparison to our earlier experiments [39], we verified safety of a larger set of benchmarks with nuXmv. We believe that this is due to the improvements in nuXmv and, probably, slight modifications of the benchmarks from [37].

NBAC and NBACC are challenging as the model checker produces many spurious counterexamples, which are an artifact of counter abstraction losing or adding processes. When using SAT-based model checking, the individual calls to nuXmv are fast, but the abstraction-refinement loop times out, due to a large number of refinements (about 500). BDD-based model checking times out when looking for a counterexample. Our new technique, preserves the number of proceses, and thus, there are no spurious counterexamples of this kind. In comparison to the general-purpose acceleration tool FAST, our tool uses less memory and is faster on the benchmarks where FAST is successful.

As predicted by the distributed algorithms literature, our tool finds counterexamples, when we relax the resilience condition. In contrast to counter abstraction, our new technique gives

Table 1 The benchmarks used in our experiments. Some benchmarks, e.g., ABA, require us to consider several cases on the parameters, which are mentioned in the column "Case". The meaning of the other columns is as follows: $|\mathcal{L}|$ is the number of local states in TA, $|\mathcal{R}|$ is the number of rules in TA, $|\Phi^{\mathrm{rise}}|$ and $|\Phi^{\mathrm{fall}}|$ is the number of (R)- and (F)-guards respectively. Finally, $|\mathcal{S}|$ is the number of enumerated schemas, and Bound is the theoretical upper bound on $|\mathcal{S}|$, as given in Theorem 4.2

#	Input	Case	Threshold Automaton				Schemas	
	FTDA	(if more than one)	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\Phi^{\mathrm{rise}}\rvert$	$\lvert\Phi^{\mathrm{fall}}\rvert$	$\lvert\mathcal{S}\rvert$	Theor. Bound
1	FRB	—	7	10	1	0	1	1
2	STRB	—	7	15	3	0	4	6
3	NBACC	—	78	1356	0	0	1	1
4	NBAC	—	77	988	6	0	448	720
5	NBACG	—	24	44	4	0	14	24
6	CF1S	$f = 0$	41	266	4	0	14	24
7	CF1S	$f = 1$	41	266	4	1	60	120
8	CF1S	$f > 1$	68	672	6	1	3429	5040
9	C1CS	$f = 0$	101	1254	8	0	70	$4 \cdot 10^4$
10	C1CS	$f = 1$	70	629	6	1	140	5040
11	C1CS	$f > 1$	101	1298	8	1	630	$3.6 \cdot 10^5$
12	BOSCO	$\lfloor \frac{n+3t}{2} \rfloor + 1 = n - t$	28	126	6	0	20	720
13	BOSCO	$\lfloor \frac{n+3t}{2} \rfloor + 1 > n - t$	40	204	8	0	70	$4 \cdot 10^4$
14	BOSCO	$\lfloor \frac{n+3t}{2} \rfloor + 1 < n - t$	32	158	6	0	20	720
15	BOSCO	$n > 5t \wedge f = 0$	82	1292	12	0	924	$4.8 \cdot 10^8$
16	BOSCO	$n > 7t$	90	1656	12	0	924	$4.8 \cdot 10^8$
17	ABA	$\frac{n+t}{2} = 2t + 1$	37	180	6	0	448	720
18	ABA	$\frac{n+t}{2} > 2t + 1$	61	392	8	0	2100	$4 \cdot 10^4$
19	CBC	$\lfloor \frac{n}{2} \rfloor < n - t \wedge f = 0$	164	1996	22	12	2	$2.9 \cdot 10^{38}$
20	CBC	$\lfloor \frac{n}{2} \rfloor = n - t \wedge f = 0$	73	442	17	12	2	$8.8 \cdot 10^{30}$
21	CBC	$\lfloor \frac{n}{2} \rfloor < n - t \wedge f > 0$	304	6799	27	12	5	$2 \cdot 10^{46}$
22	CBC	$\lfloor \frac{n}{2} \rfloor = n - t \wedge f > 0$	161	2040	22	12	5	$2.9 \cdot 10^{38}$

us concrete values of the parameters and shows how many processes move at each step of the counterexample.

Our new method uses integer counters and thus does not introduce spurious behavior due to counter abstraction, but still has spurious behavior due to data abstraction on complex FTDAs such as BOSCO, C1CS, and NBAC. In these cases, we manually refine the interval domain by adding new symbolic interval borders, see [33]. We believe that these intervals can be obtained directly from threshold automata, and no refinement is necessary. We leave this question to future work.

Sets of schemas and time to check a single schema On one hand, Theorem 4.2 gives us a theoretical bound on the number of schemas to be explored. On the other hand, optimizations discussed in Sect. 8 introduce many ways of reducing the number of schemas. Two columns in Table 1 compare the theoretical bound and the practical number of schemas: the column "Theoretical bound" shows the bound of $(|\Phi^{\mathrm{rise}}| + |\Phi^{\mathrm{fall}}|)!$, while the column $|\mathcal{S}|$ shows the actual number of schemas. (For reachability, we are merging the schemas with the prefix tree, and thus the actual number of explored schemas is even smaller.) As one can see, the theoretical bound is quite pessimistic, and is only useful to show completeness of the set of

Table 2 Summary of our experiments on AMD Opteron® 6272, 32 cores, 192 GB. The symbols are: "⏱" for timeout (72 h. for BDD and 24 h. otherwise); "💣" for memory overrun of 32 GB; "⚠" for BDD nodes overrun; "↻" for timeout in the refinement loop (72 h. for BDD and 24 h. otherwise); "☹" for spurious counterexamples due to counter abstraction

#	Input FTDA	Time, seconds PARA[2]	Time, seconds FAST	Time, seconds BMC	Time, seconds BDD	Memory, GB PARA[2]	Memory, GB FAST	Memory, GB BMC	Memory, GB BDD
1	**FRB**	1	1	1	1	0.1	0.1	0.1	0.1
2	**STRB**	1	1	3	2	0.1	0.1	0.1	0.1
3	**NBACC**	13	💣	⏱	⏱	0.1	💣	⏱	⏱
4	**NBAC**	88	💣	↻	⏱	0.1	💣	↻	⏱
5	**NBACG**	1	⚠	☹	↻	0.1	⚠	☹	↻
6	**CF1S**	6	2227	723	122	0.1	10.7	1.5	0.2
7	**CF1S**	11	6510	2235	2643	0.1	22.1	2.0	0.4
8	**CF1S**	263	⚠	⏱	40451	0.3	⚠	⏱	1.9
9	**C1CS**	45	💣	💣	10071	0.1	💣	💣	2.5
10	**C1CS**	21	💣	94962	87141	0.1	💣		9.3
11	**C1CS**	171	💣	💣	⏱	0.3	💣	💣	⏱
12	**BOSCO**	3	⚠	17892	294	0.1	⚠	1.4	0.2
13	**BOSCO**	17	⚠	↻	☹	0.1	⚠	↻	☹
14	**BOSCO**	5	⚠	2424	4	0.1	⚠	1.9	0.1
15	**BOSCO**	1013	💣	⏱	405	0.2	💣	⏱	0.7
16	**BOSCO**	1459	⚠	⏱	847	0.4	⚠	⏱	1.3
17	**ABA**	16	767	☹	11	0.1	3.5	☹	0.1
18	**ABA**	294	5757	☹	41	0.3	12.4	☹	0.2
19	**CBC**	128	💣	💣	💣	0.6	💣	💣	💣
20	**CBC**	9	⚠	2671	41873	0.1	⚠	2.8	9.9
21	**CBC**	3351	3304	💣	💣	19.3	0.1	💣	💣
22	**CBC**	215	⚠	💣	💣	4.0	⚠	💣	💣

schemas. The much smaller numbers for the fault-tolerant distributed algorithms are due to a natural order on guards, e.g., as $x \geq t + 1$ becomes true earlier than $x \geq n - t$ under the resilience condition $n > 3t$. The drastic reduction in the case of CBC is due to the control flow optimization discussed in Sect. 8 and the fact that basically all guards are forward-unlocking.

When doing experiments, we noticed that the only kinds of guards that cannot be treated by our optimizations and blow up the number of schemas are the guards that use independent shared variables. For instance, consider the guards $x_0 \geq n - t$ and $x_1 \geq n - t$ that are counting the number of 0's and 1's sent by the correct processes. Even though they are mutually exlusive under the resilience condition $n > 3t$, our tool has to explore all possible orderings of these guards. We are not aware of a reduction that would prevent our method from exploding in the number of schemas for this example.

Since the schemas can be checked independently, one can check them in parallel. Figure 9 shows a distribution of schemas along with the time needed to check an individual schema. There are only a few divergent schemas that required more than 7 s to get checked, while the large portion of schemas require 1–3 s. Hence, a parallel implementation of the tool should verify the algorithms significantly faster. We leave such a parallel extension for future work.

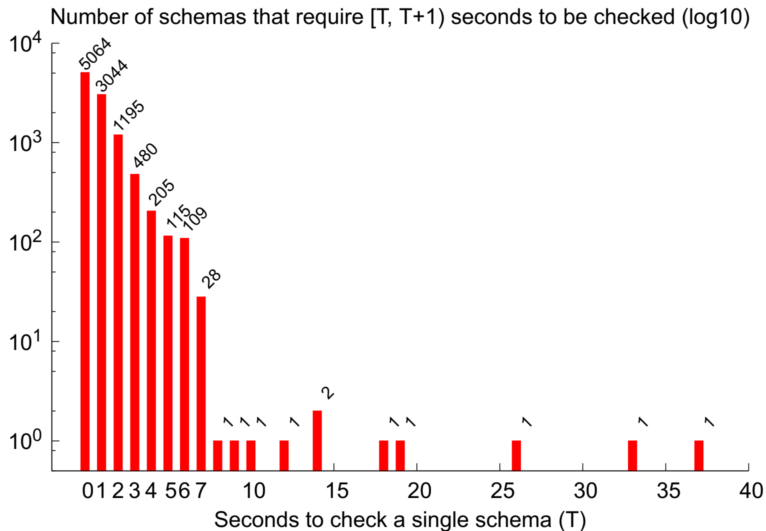


Fig. 9 The times required to check individual schemas and the distribution of schemas over these times (the value 0 refers to the running times of less than a second). The benchmarks containing the schemas that are verified in (a) $T \geq 8$ sec. and (b) $T \geq 18$ sec. are: (a) C1CS, CBC, CF1S, and (b) CBC and CF1S

11 Discussions and related work

We introduced a method to efficiently check reachability properties of FTDAs in a parameterized way. If $n > 7t$ as for BOSCO, even the simplest interesting case with $t = 2$ leads to a system size that is out of range of explicit state model checking. Hence, FTDAs force us to develop parameterized verification methods.

The problem we consider is concerned with parameterized model checking, for which many interesting results exist [14,15,21–23,35]; cf. [7] for a survey. However, the FTDAs considered by us run under the different assumptions.

From a methodological viewpoint, our approach combines techniques from several areas including compact programs [49], counter abstraction [4,55], completeness thresholds for bounded model checking [6,16,42], partial order reduction [8,28,53,59], and Lipton's movers [48]. Regarding counter automata, our result entails *flattability* [46] of every counter system of threshold automata: a complete set of schemas immediately gives us a flat counter automaton. Hence, the acceleration-based semi-algorithms [3,46] should in principle terminate on the systems of TAs, though it did not always happen in our experiments. Similar to our SMT queries based on schemas, the *inductive data flow graphs* iDFG introduced in [24] are a succinct representations of schedules (they call them traces) for systems where the number of processes (or threads) is fixed. The work presented in [25] then considers parameterized verification. Further, our execution schemas are inspired by a general notion of *semi-linear path schemas* SLPS [45,46]. We construct a small complete set of schemas and thus a provably small SLPS. Besides, we distinguish counter systems and counter abstraction: the former counts processes as integers, while the latter uses counters over a finite abstract domain, e.g., $\{0, 1, \mathit{many}\}$ [55].

Many distributed algorithms can be represented with I/O Automata [50] or TLA+ [44]. In these frameworks, correctness is typically shown with a proof assistant, while model checking

is used as a debugger on small instances. Parameterized model checking is not a concern there, except one notable result [32].

The results presented in this article can be used to check reachability properties of FTDAs. We can thus establish safety of FTDAs. However, for fault-tolerant distributed algorithms liveness is as important as safety: The seminal impossibility result by Fischer, Lynch, and Paterson [26] states that a fault-tolerant consensus algorithm cannot ensure both safety and liveness in asynchronous systems. In recent work [37] we also considered liveness verification, or more precisely, verification of temporal logic specification with the G and F temporal operators. In [37], we use the results of this article as a black box and show that combinations of schemas can be used to generate counterexamples to liveness properties, and that we can verify both safety and liveness by complete SMT-based bounded model checking.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF). We are grateful to Azadeh Farzan for valuable discussions during her stay in Vienna and to the anonymous reviewers for their insightful comments regarding partial order reduction, and for suggestions that helped us in improving the presentation of the paper.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Attiya H, Welch J (2004) Distributed computing, 2nd edn. Wiley, New York
2. ByMC: Byzantine model checker (2013). http://forsyte.tuwien.ac.at/software/bymc/. Accessed Dec 2016
3. Bardin S, Finkel A, Leroux J, Petrucci L (2008) Fast: acceleration from theory to practice. STTT 10(5):401–424
4. Basler G, Mazzucchi M, Wahl T, Kroening D (2009) Symbolic counter abstraction for concurrent software. In: CAV. LNCS, vol 5643, pp 64–78
5. Biere A (2013) Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In: Proceedings of SAT competition 2013; Solver and p. 51
6. Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: TACAS. LNCS, vol 1579, pp 193–207
7. Bloem R, Jacobs S, Khalimov A, Konnov I, Rubin S, Veith H, Widder J (2015) Decidability of parameterized verification, synthesis lectures on distributed computing theory. Morgan & Claypool, San Rafael
8. Bokor P, Kinder J, Serafini M, Suri N (2011) Efficient model checking of fault-tolerant distributed protocols. In: DSN, pp 73–84
9. Bracha G, Toueg S (1985) Asynchronous consensus and broadcast protocols. J ACM 32(4):824–840
10. Brasileiro FV, Greve F, Mostéfaoui A, Raynal M (2001) Consensus in one communication step. In: PaCT. LNCS, vol 2127, pp 42–50
11. Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S (2014) The nuXmv symbolic model checker. In: CAV. LNCS, vol 8559, pp 334–342
12. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267
13. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794
14. Clarke E, Talupur M, Touili T, Veith H (2004) Verification by network decomposition. In: CONCUR 2004, vol 3170, pp 276–291
15. Clarke E, Talupur M, Veith H (2008) Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In: TACAS'08/ETAPS'08. Springer, Berlin, pp 33–47
16. Clarke EM, Kroening D, Ouaknine J, Strichman O (2004) Completeness and complexity of bounded model checking. In: VMCAI. LNCS, vol 2937, pp 85–96
17. De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Tools and algorithms for the construction and analysis of systems. LNCS, vol 1579, pp 337–340

18. Dobre D, Suri N (2006) One-step consensus with zero-degradation. In: DSN, pp 137–146
19. Drăgoi C, Henzinger TA, Zufferey D (2016) PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL, pp 400–415
20. Drăgoi C, Henzinger TA, Veith H, Widder J, Zufferey D (2014) A logic-based framework for verifying consensus algorithms. In: VMCAI. LNCS, vol 8318, pp 161–181
21. Emerson E, Namjoshi K (1995) Reasoning about rings. In: POPL, pp 85–94
22. Emerson EA, Kahlon V (2003) Model checking guarded protocols. In: LICS. IEEE, pp 361–370
23. Esparza J, Ganty P, Majumdar R (2013) Parameterized verification of asynchronous shared-memory systems. In: CAV, pp 124–140
24. Farzan A, Kincaid Z, Podelski A (2013) Inductive data flow graphs. In: POPL, pp 129–142
25. Farzan A, Kincaid Z, Podelski A (2015) Proof spaces for unbounded parallelism. In: POPL, pp 407–420
26. Fischer MJ, Lynch NA, Paterson MS (1985) Impossibility of distributed consensus with one faulty process. J ACM 32(2):374–382
27. Gmeiner A, Konnov I, Schmid U, Veith H, Widder J (2014) Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In: SFM. LNCS, vol 8483. Springer, Berlin, pp 122–171
28. Godefroid P (1990) Using partial orders to improve automatic verification methods. In: CAV. LNCS, vol 531, pp 176–185
29. Guerraoui R (2002) Non-blocking atomic commit in asynchronous distributed systems with failure detectors. Distrib Comput 15(1):17–25
30. https://github.com/konnov/fault-tolerant-benchmarks/tree/master/fmsd17
31. Hawblitzel C, Howell J, Kapritsos M, Lorch JR, Parno B, Roberts ML, Setty STV, Zill B (2015) Ironfleet: proving practical distributed systems correct. In: SOSP, pp 1–17
32. Jensen H, Lynch N (1998) A proof of Burns n-process mutual exclusion algorithm using abstraction. In: Steffen B (ed) TACAS. LNCS, vol 1384. Springer, Berlin, pp 409–423
33. John A, Konnov I, Schmid U, Veith H, Widder J (2013) Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD, pp 201–209
34. John A, Konnov I, Schmid U, Veith H, Widder J (2013) Towards modeling and model checking fault-tolerant distributed algorithms. In: SPIN. LNCS, vol 7976, pp 209–226
35. Kaiser A, Kroening D, Wahl T (2012) Efficient coverability analysis by proof minimization. In: CONCUR, pp 500–515
36. Kesten Y, Pnueli A (2000) Control and data abstraction: the cornerstones of practical formal verification. STTT 2:328–342
37. Konnov I, Lazić M, Veith H, Widder J (2017) A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL, pp 719–734
38. Konnov I, Veith H, Widder J (2014) On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. In: CONCUR. LNCS, vol 8704, pp 125–140
39. Konnov I, Veith H, Widder J (2015) SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In: CAV (Part I). LNCS, vol 9206, pp 85–102
40. Konnov I, Veith H, Widder J (2016) What you always wanted to know about model checking of fault-tolerant distributed algorithms. In: PSI 2015, revised selected papers. LNCS, vol 9609. Springer, pp 6–21
41. Konnov I, Veith H, Widder J (2017) On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. Inf Comput 252:95–109
42. Kroening D, Strichman O (2003) Efficient computation of recurrence diameters. In: VMCAI. LNCS, vol 2575, pp 298–309
43. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
44. Lamport L (2002) Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co. Inc, Boston
45. Leroux J, Sutre G (2004) On flatness for 2-dimensional vector addition systems with states. In: CONCUR 2004-concurrency theory. Springer, pp 402–416
46. Leroux J, Sutre G (2005) Flat counter automata almost everywhere! In: ATVA. LNCS, vol 3707, pp 489–503
47. Lesani M, Bell CJ, Chlipala A (2016) Chapar: certified causally consistent distributed key-value stores. In: POPL, pp 357–370
48. Lipton RJ (1975) Reduction: a method of proving properties of parallel programs. Commun ACM 18(12):717–721
49. Lubachevsky BD (1984) An approach to automating the verification of compact parallel coordination programs. I. Acta Inform 21(2):125–169
50. Lynch N (1996) Distributed algorithms. Morgan Kaufman, Burlington

51. Mostéfaoui A, Mourgaya E, Parvédy PR, Raynal M (2003) Evaluating the condition-based approach to solve consensus. In: DSN, pp 541–550
52. Padon O, McMillan KL, Panda A, Sagiv M, Shoham S (2016) Ivy: safety verification by interactive generalization. In: PLDI, pp 614–630
53. Peled D (1993) All from one, one for all: on model checking using representatives. In: CAV. LNCS, vol 697, pp 409–423
54. Peluso S, Turcu A, Palmieri R, Losa G, Ravindran B (2016) Making fast consensus generally faster. In: DSN, pp 156–167
55. Pnueli A, Xu J, Zuck L (2002) Liveness with (0,1,∞)-counter abstraction. In: CAV. LNCS, vol 2404, pp 93–111
56. Raynal M (1997) A case study of agreement problems in distributed systems: non-blocking atomic commitment. In: HASE, pp 209–214
57. Song YJ, van Renesse R (2008) Bosco: one-step Byzantine asynchronous consensus. In: DISC. LNCS, vol 5218, pp 438–450
58. Srikanth T, Toueg S (1987) Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distrib Comput 2:80–94
59. Valmari A (1991) Stubborn sets for reduced state space generation. In: Advances in Petri Nets 1990. LNCS, vol 483. Springer, pp 491–515
60. Wilcox JR, Woos D, Panchekha P, Tatlock Z, Wang X, Ernst MD, Anderson TE (2015) Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI, pp 357–368

Chapter 6

A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms

Igor Konnov, Marijana Lazić, Helmut Veith, Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. ACM POPL, pp. 719–734, 2017.

URL: http://dl.acm.org/citation.cfm?id=3009860

A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant Distributed Algorithms

Igor Konnov Marijana Lazić Helmut Veith * Josef Widder

TU Wien, Austria

{konnov, lazic, veith, widder}@forsyte.at

Abstract

Distributed algorithms have many mission-critical applications ranging from embedded systems and replicated databases to cloud computing. Due to asynchronous communication, process faults, or network failures, these algorithms are difficult to design and verify. Many algorithms achieve fault tolerance by using threshold guards that, for instance, ensure that a process waits until it has received an acknowledgment from a majority of its peers. Consequently, domain-specific languages for fault-tolerant distributed systems offer language support for threshold guards.

We introduce an automated method for model checking of safety and liveness of threshold-guarded distributed algorithms in systems where the number of processes and the fraction of faulty processes are parameters. Our method is based on a *short counterexample property*: if a distributed algorithm violates a temporal specification (in a fragment of LTL), then there is a counterexample whose length is bounded and independent of the parameters. We prove this property by (i) characterizing executions depending on the structure of the temporal formula, and (ii) using commutativity of transitions to accelerate and shorten executions. We extended the ByMC toolset (Byzantine Model Checker) with our technique, and verified liveness and safety of 10 prominent fault-tolerant distributed algorithms, most of which were out of reach for existing techniques.

Categories and Subject Descriptors F.3.1 [*Logic and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.4.5 [*Software*]: Operating systems: Fault-tolerance, Verification

Keywords Parameterized model checking, Byzantine faults, fault-tolerant distributed algorithms, reliable broadcast

* We dedicate this article to the memory of Helmut Veith, who passed away tragically after we finished the first draft together. In addition to contributing to this work, Helmut initiated our long-term research program on verification of fault-tolerant distributed algorithms, which made this paper possible.

Supported by: the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403 and S11405), project PRAVDA (P27722), and Doctoral College LogiCS (W1255-N23); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009860

1. Introduction

Distributed algorithms have many applications in avionic and automotive embedded systems, computer networks, and the internet of things. The central idea is to achieve dependability by replication, and to ensure that all correct replicas behave as one, even if some of the replicas fail. In this way, the correct operation of the system is more reliable than the correct operation of its parts. Fault-tolerant algorithms typically have been used in applications where highest reliability is required because human life is at risk (e.g., automotive or avionic industries), and even unlikely failures of the system are not acceptable. In contrast, in more mainstream applications like replicated databases, human intervention to restart the system from a checkpoint was often considered to be acceptable, so that expensive fault tolerance mechanisms were not used in conventional applications. However, new application domains such as cloud computing provide a new motivation to study fault-tolerant algorithms: with the huge number of computers involved, faults are the norm [53] rather than an exception, so that fault tolerance becomes an economic necessity; and so does the correctness of fault tolerance mechanisms. Hence, design, implementation, and verification of distributed systems constitutes an active research area [7, 23, 41, 42, 48, 57, 67]. Although distributed algorithms show complex behavior, and are difficult to understand for human engineers, there is only very limited tool support to catch logical errors in fault-tolerant distributed algorithms at design time.

The state of the art in the design of fault-tolerant systems is exemplified by the recent work on Paxos-like distributed algorithms like Raft [54] or M^{2}PAXOS [57]. The designers encode these algorithms in TLA+ [65], and use the TLC model checker to automatically find bugs in small instances, i.e., in distributed systems containing, e.g., three processes. Large distributed systems (e.g., clouds) need guarantees for *all* numbers of processes. These guarantees are typically given using hand-written mathematical proofs. In principle, these proofs could be encoded and machine-checked using the TLAPS proof system [16], PVS [49], Isabelle [15], Coq [48], Nuprl [60], or similar systems; but this requires human expertise in the proof checkers and in the application domain, and a lot of effort.

Ensuring correctness of the implementation is an open challenge: As the implementations are done by hand [54, 57], the connection between the specification and the implementation is informal, such that there is no formal argument about the correctness of the implementation. To address the discrepancy between design, implementation, and verification, Drăgoi et al. [23] introduced a domain-specific language PSync which is used for two purposes: (i) it compiles into running code, and (ii) it is used for verification. Their verification approach [24], requires a developer to provide invariants, and similar verification conditions. While this approach requires less human intervention than writing machine-checkable proofs, coming up with invariants of distributed systems requires considerable

```
case class EchoMsg extends Message

class ReliableBroadcastOnce
      extends DSLProtocol {
 val n = ALL.size          // nr. processes
 val t = ALL.size / 3 − 1  // max. faults
 var accept: Boolean = False

 UPON RECEIVING START WITH v DO {
  IF v == 1 THEN // check the initial value
  SEND EchoMsg TO ALL
 }
 UPON RECEIVING EchoMsg TIMES t + 1 DO {
  SEND EchoMsg TO ALL // >= 1 correct
 }
 UPON RECEIVING EchoMsg TIMES n − t DO {
  accept = True // almost all correct
 }
}
```

Figure 1. Code example of a distributed algorithm in DISTAL [7]. A distributed system consists of n processes, at most $t < n/3$ of which are Byzantine faulty. The correct ones execute the code, and no assumptions is made about the faulty processes.

human ingenuity. The Mace [41] framework is based on a similar idea, and is an extension to C++. While being fully automatic, their approach to correctness is light-weight in that it uses a tool that explores random walks to find (not necessarily all) bugs, rather than actually verifying systems.

In this paper we focus on automatic verification methods for programming constructs that are typical for fault-tolerant distributed algorithms. Figure 1 is an example of a distributed algorithm in the domain-specific language DISTAL [7]. It encodes the core of the reliable broadcast protocol from [64], which is used as building block of many fault-tolerant distributed systems. Line 13 and Line 16 use so-called "threshold guards" that check whether a given number of messages from distinct senders arrived at the receiver. As threshold guards are the central algorithmic idea for fault tolerance, domain-specific languages such as DISTAL or PSync have constructs for them (see [23] for an overview of domain-specific languages and formalization frameworks for distributed systems). For instance, the code in Figure 1 works for systems with n processes among which t can fail, with $t < n/3$ as required for Byzantine fault tolerance [56]. In such systems, waiting for messages from $n - t$ processes ensures that if all correct processes send messages, then faulty processes cannot prevent progress. Similarly, waiting for $t + 1$ messages ensures that at least one message was sent by a correct process. Konnov et al. [42] introduced an automatic method to verify safety of algorithms with threshold guards. Their method is parameterized in that it verifies distributed algorithms for all values of parameters (n and t) that satisfy a resilience condition ($t < n/3$). This work bares similarities to the classic work on reduction for parallel programs by Lipton [50]. Lipton proves statements like "all P operations on a semaphore are left movers with respect to operations on other processes." He proves that given a run that ends in a given state, the same state is reached by the run in which the P operation has been moved. Konnov et al. [42] do a similar analysis for threshold-guarded operations, in which they analyze the relation between statements from Figure 1 like "`send EchoMsg`" and "`UPON RECEIVING EchoMsg TIMES t + 1`" in order to determine which statements are movable. From this, they develop an offline partial order reduction that together with acceleration [6, 44] reduced reachability checking to complete bounded model checking using SMT. In this way, they automatically check safety of fault-tolerant algorithms.

However, for fault-tolerant distributed algorithms liveness is as important as safety: This comes from the celebrated impossibility result by Fischer, Lynch, and Paterson [32] that states that a fault-tolerant consensus algorithm cannot ensure both safety and liveness in asynchronous systems. It is folklore that designing a safe fault-tolerant distributed algorithm is trivial: *just do nothing*; e.g., by never committing transactions, one cannot commit them in inconsistent order. Hence, a technique that verifies only safety may establish the "correctness" of a distributed algorithm that never does anything useful. To achieve trust in correctness of a distributed algorithm, we need tools that verify both safety and liveness.

As exemplified by [31], liveness verification of parameterized distributed and concurrent systems is still a research challenge. Classic work on parameterized model checking by German and Sistla [35] has several restrictions on the specifications ($\forall i.\, \phi(i)$) and the computational model (rendezvous), which are incompatible with fault-tolerant distributed algorithms. In fact, none of the approaches (e.g., [18, 26, 27, 59]) surveyed in [9] apply to the algorithms we consider. More generally, in the parameterized case, going from safety to liveness is not straightforward. There are systems where safety is decidable and liveness is not [28].

Contributions. We generalize the approach by Konnov et al. [42, 44] to liveness by presenting a framework and a model checking tool that takes as input a description of a distributed algorithm (in our variant [36] of Promela [39]) and specifications in a fragment of linear temporal logic. It then shows correctness for all parameter values (e.g., n and t) that satisfy the required resilience condition (e.g., $t < n/3$), or reports a counterexample:

1. As in the classic result by Vardi and Wolper [66], we observe that it is sufficient to search for counterexamples that have the form of a lasso, i.e., after a finite prefix an infinite loop is entered. Based on this, we analyze specifications automatically, in order to enumerate possible shapes of lassos depending on temporal operators **F** and **G** and evaluations of threshold guards.
2. We automatically do offline partial order reduction using the algorithm's description. For this, we introduce a more refined mover analysis for threshold guards and temporal properties. We extend Lipton's reduction method [50] (re-used and extended by many others [19, 22, 25, 34, 44, 47]), so that we maintain invariants, which allows us to go beyond reachability and verify specifications with the temporal operators **F** and **G**.
3. By combining acceleration [6, 44] with Points 1 and 2, we obtain a short counterexample property, that is, that infinite executions (which may potentially be counterexamples) have "equivalent" representatives of bounded length. The bound depends on the process code and is independent of the parameters. The equivalence is understood in terms of temporal logic specifications that are satisfied by the original executions and the representatives, respectively. We show that the length of the representatives increases mildly compared to reachability checking in [42]. This implies a so-called completeness threshold [46] for threshold-based algorithms and our fragment of LTL.
4. Consequently, we only have to check a reasonable number of SMT queries that encode parameterized and bounded-length representatives of executions. We show that if the parameterized system violates a temporal property, then SMT reports a counterexample for one of the queries. We prove that otherwise the specification holds for all system sizes.
5. Our theoretical results and our implementation push the boundary of liveness verification for fault-tolerant distributed algorithms. While prior results [40] scale just to two out of ten benchmarks from [42], we verified safety and liveness of all ten. These benchmarks originate from distributed algorithms [11, 12, 14, 21, 37, 52, 61, 63, 64] that constitute the core of important services such as replicated state machines.

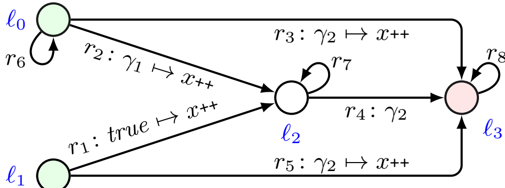


Figure 2. The threshold automaton corresponding to Figure 1 with $\gamma_1 : x \geq (t+1) - f$ and $\gamma_2 : x \geq (n-t) - f$ over parameters n, t, and f, representing the number of processes, the upper bound on the faulty processes (used in the code), and the actual number of faulty processes. The negative number $-f$ in the threshold is used to model the environment, and captures that at most f of the received messages may have been sent by faulty processes.

From a theoretical viewpoint, we introduce new concepts and conduct extensive proofs (the proofs can be found in [43]) for Points 1 and 2. From a practical viewpoint, we have built a complete framework for model checking of fault-tolerant distributed algorithms that use threshold guards, which constitute the central programming paradigm for dependable distributed systems.

2. Representation of Distributed Algorithms

2.1 Threshold Automata

As internal representation in our tool, and in the theoretical work of this paper, we use *threshold automata* (TA) defined in [44]. The TA that corresponds to the DISTAL code from Figure 1 is given in Figure 2. The threshold automaton represents the local control flow of a single process, where arrows represent local transitions that are labeled with $\varphi \mapsto \texttt{act}$: Expression φ is a threshold guard and the action act may increment a shared variable.

Example 2.1. The TA from Figure 2 is quite similar to the code in Figure 1: if `START` is called with $v = 1$ this corresponds to the initial local state ℓ_1, while otherwise a process starts in ℓ_0. Initially a process has not sent any messages. The local state ℓ_2 in Figure 2 captures that the process has sent `EchoMsg` and `accept` evaluates to false, while ℓ_3 captures that the process has sent `EchoMsg` and `accept` evaluates to true. The syntax of Figure 1, although checking how many messages of some type are received, hides bookkeeping details and the environment, e.g., message buffers. For our verification technique, we need to make such issues explicit: The shared variable x stores the number of correct processes that have sent `EchoMsg`. Incrementing x models that `EchoMsg` is sent when the transition is taken. Then, execution of Line 9 corresponds to the transition r_1. Executing Line 13 is captured by r_2: the check whether $t+1$ messages are received is captured by the fact that r_2 has the guard γ_1, that is, $x \geq (t+1) - f$. Intuitively, this guard checks whether sufficiently many processes have sent `EchoMsg` (i.e., increased x), and takes into account that at most f messages may have been sent by faulty processes. Namely, if we observe the guard in the equivalent form $x + f \geq t+1$, then we notice that it evaluates to true when the total number of received `EchoMsg` messages from correct processes (x) and potentially received messages from faulty processes (at most f), is at least $t+1$, which corresponds to the guard of Line 13. Transition r_4 corresponds to Line 16, r_3 captures that Line 9 and Line 16 are performed in one protocol step, and r_5 captures Line 13 and Line 16. ◁

While the example shows that the code in a domain-specific language and a TA are quite close, it should be noted that in reality, things are slightly more involved. For instance, the DISTAL runtime takes care of the bookkeeping of sent and received messages (waiting queues at different network layers, buffers, etc.), and just triggers the high-level protocol when a threshold guard evaluates to true. This typically requires counting the number of received messages. While these local counters are present in the implementation, they are abstracted in the TA. For the purpose of this paper we do not need to get into the details. Discussions on data abstraction and automated generation of TAs from code similar to DISTAL can be found in [45].

We recall the necessary definitions introduced in [44]. A threshold automaton is a tuple $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ whose components are defined as follows: The *local states* and the *initial states* are in the finite sets $\mathcal{L}$ and $\mathcal{I} \subseteq \mathcal{L}$, respectively. For simplicity, we identify local states with natural numbers, i.e., $\mathcal{L} = \{1, \ldots, |\mathcal{L}|\}$. *Shared variables* and *parameter variables* range over $\mathbb{N}_0$ and are in the finte sets Γ and Π, respectively. The *resilience condition RC* is a formula over parameter variables in linear integer arithmetic, and the *admissible parameters* are $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : \mathbf{p} \models RC\}$. After an example for resilience conditions, we will conclude the definition of a threshold automaton by defining $\mathcal{R}$ as the finite set of rules.

Example 2.2. The admissible parameters and resilience conditions are motivated by fault-tolerant distributed algorithms: Let n be the number of processes, t be the assumed number of faulty processes, and in a run, f be the actual number of faults. For these parameters, the famous result by Pease, Shostak and Lamport [56] states that agreement can be solved iff the resilience condition $n > 3t \wedge t \geq f \geq 0$ is satisfied. Given such constraints, the set $\mathbf{P}_{RC}$ is infinite, and in Section 2.2 we will see that this results in an infinite state system. ◁

A *rule* is a tuple $(\mathit{from}, \mathit{to}, \varphi^{\leq}, \varphi^{>}, \mathbf{u})$, where *from* and *to* are from $\mathcal{L}$, and capture from which local state to which a process moves via that rule. A rule can only be executed if $\varphi^{\leq}$ and $\varphi^{>}$ are true; both are conjunction of guards. Each guard consists of a shared variable $x \in \Gamma$, coefficients $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, and parameter variables $p_1, \ldots, p_{|\Pi|} \in \Pi$ so that $x \geq a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i$ is a *lower guard* and $x < a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i$ is an *upper guard*. Then, Φ^{rise} and Φ^{fall} are the sets of lower and upper guards.[1] Rules may increase shared variables using an update vector $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$ that is added to the vector of shared variables. Finally, $\mathcal{R}$ is the finite set of rules.

Example 2.3. A rule corresponds to an edge in Figure 2. The pair $(\mathit{from}, \mathit{to})$ encodes the edge while $(\varphi^{\leq}, \varphi^{>}, \mathbf{u})$ encodes the edge label. For example, rule r_2 would be $(\ell_0, \ell_2, \gamma_1, \top, 1)$. Thus, a rule corresponds to a (guarded) statement from Figure 1 (or combined statements as discussed in Example 2.1). ◁

The above definition of TAs is quite general. It allows loops, increase of shared variables in loops, etc. As has been observed in [44], if one does not restrict increases on shared variables, the resulting systems may produce runs that visit infinitely many states, and there is little hope for a complete verification method. Hence, Konnov et al. [42] analyzed the TAs of the benchmarks [11, 12, 14, 21, 37, 52, 61, 63, 64]: They observed that some states have self-loops (corresponding to busy-waiting for messages to arrive) and in the case of failure detector based algorithms [61] there are loops that consist of at most two rules. None of the rules in loops increase shared variables. In our theory, we allow more general TAs than actually found in the benchmarks. In more detail, we make the following assumption:

[1] Compared to [42], we use the more intuitive notation of Φ^{rise} and Φ^{fall}: lower guards can only change from false to true (rising), while upper guards can only change from true to false (falling); cf. Proposition 5.1.

Threshold automata for fault-tolerant distributed algorithms. As in [44], we assume that if a rule r is in a loop, then $r.\mathbf{u} = \mathbf{0}$. In addition, we use the restriction that all the cycles of a TA are simple, i.e., between any two locations in a cycle there exists exactly one node-disjoint directed path (nodes in cycles may have self-loops). We conjecture that this restriction can be relaxed as in [42], but this is orthogonal to our work.

Example 2.4. In the TA from Figure 2 we use the shared variable x as the number of correct processes that have sent a message. One easily observes that the rules that update x do not belong to loops. Indeed, all the benchmarks [11, 12, 14, 21, 37, 52, 61, 63, 64] share this structure. This is because at the algorithmic level, all these algorithms are based on the reliable communication assumption (no message loss and no spurious message generation/duplication), and not much is gained by resending the same message. In these algorithms a process checks whether sufficiently many processes (e.g., a majority) have sent a message to signal that they are in some specific local state. Consequently, a receiver would ignore duplicate messages from the same sender. In our analysis we exploit this characteristic of distributed algorithms with threshold guards, and make the corresponding assumption that processes do not send (i.e., increase x) from within a loop. Similarly, as a process cannot make the sending of a message undone, we assume that shared variables are never decreased. So, while we need these assumptions to derive our results, they are justified by our application domain. ◁

2.2 Counter Systems

A threshold automaton models a single process. Now the question arises how we define the composition of multiple processes that will result in a distributed system. Classically, this is done by parallel composition and interleaving semantics: A state of a distributed system that consists of n processes is modeled as n-dimensional vector of local states. The transition to a successor state is then defined by non-deterministically picking a process, say i, and changing the ith component of the n-dimensional vector according to the local transition relation of the process. However, for our domain of threshold-guarded algorithms, we do not care about the precise n-dimensional vector so that we use a more efficient encoding: It is well-known that the system state of specific distributed or concurrent systems can be represented as a counter system [2, 44, 51, 59]: instead of recording for some local state ℓ, which processes are in ℓ, we are only interested in *how many processes are in* ℓ. In this way, we can efficiently encode transition systems in SMT with linear integer arithmetics. Therefore, we formalize the semantics of the threshold automata by counter systems.

Fix a threshold automaton TA, a function (expressible as linear combination of parameters) $N\colon \mathbf{P}_{RC} \to \mathbb{N}_0$ that determines the number of modeled processes, and admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$. A counter system $\mathsf{Sys}(\mathsf{TA})$ is defined as a transition system (Σ, I, R), with configurations Σ and I and transition relation R defined below.

Definition 2.5. *A configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ consists of a vector of* counter values $\sigma.\boldsymbol{\kappa} \in \mathbb{N}_0^{|\mathcal{L}|}$, *a vector of* shared variable values $\sigma.\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$, *and a vector of* parameter values $\sigma.\mathbf{p} = \mathbf{p}$. *The set Σ contains all configurations. The initial configurations are in set I, and each initial configuration σ satisfies $\sigma.\mathbf{g} = \mathbf{0}$, $\sum_{i\in\mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = N(\mathbf{p})$, and $\sum_{i\notin\mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = 0$.*

Example 2.6. The safety property from Example 2.2, refers to an initial configuration that satisfies resilience condition $n > 3t \wedge t \geq f \geq 0$, e.g., $4 > 3 \cdot 1 \wedge 1 \geq 0 \geq 0$ such that $\sigma.\mathbf{p} = (4, 1, 0)$. In our encodings we typically have N is the function $(n, t, f) \mapsto n - f$. Further, $\sigma.\boldsymbol{\kappa}[\ell_0] = N(\mathbf{p}) = n - f = 4$ and $\sigma.\boldsymbol{\kappa}[\ell_i] = 0$, for $\ell_i \in \mathcal{L} \setminus \{\ell_0\}$, and the shared variable $\sigma.\mathbf{g} = 0$. ◁

A *transition* is a pair $t = (\mathit{rule}, \mathit{factor})$ of a rule and a non-negative integer called the *acceleration factor*. For $t = (\mathit{rule}, \mathit{factor})$ we write $t.\mathbf{u}$ for $\mathit{rule}.\mathbf{u}$, etc. A transition t is *unlocked* in σ if $\forall k \in \{0, \ldots, t.\mathit{factor} - 1\}.\ (\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + k \cdot t.\mathbf{u}, \sigma.\mathbf{p}) \models t.\varphi^{\leq} \wedge t.\varphi^{>}$. A transition t is *applicable (or enabled)* in σ, if it is unlocked, and $\sigma.\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$, or $t.\mathit{factor} = 0$.

Example 2.7. This notion of applicability contains acceleration and is central for our approach. Intuitively, the value of the factor corresponds to how many times the rule is executed by different processes. In this way, we can subsume steps by an arbitrary number of processes into one transition. Consider Figure 2. If for some k, k processes are in location ℓ_1, then in classic modeling it takes k transitions to move these processes one-by-one to ℓ_2. With acceleration, however, these k processes can be moved to ℓ_2 in one step, independently of k. In this way, the bounds we compute will be independent of the parameter values. However, assuming x to be a shared variable and f being a parameter that captures the number of faults, our (crash-tolerant) benchmarks include rules like "$x < f \mapsto x{+}{+}$" for local transition to a special "crashed" state. The above definition ensures that at most $f - x$ of these transitions are accelerated into one transition (whose factor thus is at most $f - x$). This precise treatment of threshold guards is crucial for fault-tolerant distributed algorithms. The central contribution of this paper is to show how acceleration can be used to shorten schedules while maintaining specific temporal logic properties. ◁

Definition 2.8. *The configuration σ' is the result of applying the enabled transition t to σ, if*

1. $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.\mathit{factor} \cdot t.\mathbf{u}$
2. $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
3. *if $t.\mathit{from} \neq t.\mathit{to}$ then $\sigma'.\boldsymbol{\kappa}[t.\mathit{from}] = \sigma.\boldsymbol{\kappa}[t.\mathit{from}] - t.\mathit{factor}$, $\sigma'.\boldsymbol{\kappa}[t.\mathit{to}] = \sigma.\boldsymbol{\kappa}[t.\mathit{to}] + t.\mathit{factor}$, and $\forall \ell \in \mathcal{L} \setminus \{t.\mathit{from}, t.\mathit{to}\}.\ \sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$.*
4. *if $t.\mathit{from} = t.\mathit{to}$ then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$.*

In this case we use the notation $\sigma' = t(\sigma)$.

Example 2.9. Let us again consider Figure 2 with $n = 4$, $t = 1$, and $f = 1$. We consider the initial configuration where $\sigma.\boldsymbol{\kappa}[\ell_1] = n - f = 3$ and $\sigma.\boldsymbol{\kappa}[\ell_i] = 0$, for $\ell_i \in \mathcal{L} \setminus \{\ell_0\}$. The guard of rule r_5, $\gamma_2\colon x \geq (n-t) - f = 2$, initially evaluates to false because $x = 0$. The guard of rule r_1 is true, so that any transition (r_1, factor) is unlocked. As $\sigma.\boldsymbol{\kappa}[\ell_1] = 3$, all transitions (r_1, factor), for $0 \leq \mathit{factor} \leq 3$ are applicable. If the transition $(r_1, 2)$ is applied to the initial configuration, we obtain that $x = 2$ so that, after the application, γ_2 evaluates to true. Then r_5 is unlocked and the transitions $(r_5, 1)$ and $(r_5, 0)$ are applicable as $\sigma.\boldsymbol{\kappa}[\ell_1] = 1$. Since γ_2 checks for greater or equal, once it becomes true it remains true. Such monotonic behavior is given for all guards, as has already been observed in [44, Proposition 7], and is a crucial property. ◁

The transition relation R is defined as follows: Transition (σ, σ') belongs to R iff there is a rule $r \in \mathcal{R}$ and a factor $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$. A *schedule* is a sequence of transitions. For a schedule τ and an index $i : 1 \leq i \leq |\tau|$, by $\tau[i]$ we denote the ith transition of τ, and by τ^i we denote the prefix $\tau[1], \ldots, \tau[i]$ of τ. A schedule $\tau = t_1, \ldots, t_m$ is *applicable* to configuration σ_0, if there is a sequence of configurations $\sigma_1, \ldots, \sigma_m$ with $\sigma_i = t_i(\sigma_{i-1})$ for $1 \leq i \leq m$. A schedule $t_1, \ldots, t_m$ where $t_i.\mathit{factor} = 1$ for $0 < i \leq m$ is called *conventional*. If there is a $t_i.\mathit{factor} > 1$, then a schedule is *accelerated*. By $\tau \cdot \tau'$ we denote the concatenation of two schedules τ and τ'.

We will reason about schedules in Section 6 for our mover analysis, which is naturally expressed by swapping neighboring transitions in a schedule. To reason about temporal logic properties, we need to reason about the configurations that are "visited" by a schedule. For that we now introduce paths.

A finite or infinite sequence $\sigma_0, t_1, \sigma_1, \dots, t_{k-1}, \sigma_{k-1}, t_k, \dots$ of alternating configurations and transitions is called a *path*, if for every transition t_i, $i \in \mathbb{N}$, in the sequence, holds that t_i is enabled in σ_{i-1}, and $\sigma_i = t_i(\sigma_{i-1})$. For a configuration σ_0 and a finite schedule τ applicable to σ_0, by $\mathsf{path}(\sigma_0, \tau)$ we denote $\sigma_0, t_1, \sigma_1, \dots, t_{|\tau|}, \sigma_{|\tau|}$ with $\sigma_i = t_i(\sigma_{i-1})$, for $1 \le i \le |\tau|$. Similarly, if τ is an infinite schedule applicable to σ_0, then $\mathsf{path}(\sigma_0, \tau)$ represents an infinite sequence $\sigma_0, t_1, \sigma_1, \dots, t_{k-1}, \sigma_{k-1}, t_k, \dots$ where $\sigma_i = t_i(\sigma_{i-1})$, for all $i > 0$.

The evaluation of the threshold guards solely defines whether certain rules are unlocked. As was discussed in Example 2.9, along a path, the evaluations of guards are monotonic. The set of upper guards that evaluate to false and lower guards that evaluate to true — called the context — changes only finitely many times. A schedule can thus be understood as an alternating sequence of schedules without context change, and context-changing transitions. We will recall the definitions of context etc. from [42] in Section 5. We say that a schedule τ is *steady* for a configuration σ, if every configuration of $\mathsf{path}(\sigma, \tau)$ has the same context.

Due to the resilience conditions and admissible parameters, our counter systems are in general infinite state. The following proposition establishes an important property for verification.

Proposition 2.10. *Every (finite or infinite) path visits finitely many configurations.*

Proof. By Definition 2.8(3), if a transition t is applied to a configuration σ, then the sum of the counters remains unchanged, that is, $\sum_{\ell \in \mathcal{L}} \sigma.\boldsymbol{\kappa}[\ell] = \sum_{\ell \in \mathcal{L}} t(\sigma).\boldsymbol{\kappa}[\ell]$. By repeating this argument, the sum of the counters remains stable in a path. By Definition 2.8(2) the parameter values also remain stable in a path.

By Definition 2.8(1), it remains to show that in each path eventually the shared variable $\mathbf{g}$ stop increasing. Let us fix a rule $r = (\mathit{from}, \mathit{to}, \varphi^{\le}, \varphi^{>}, \mathbf{u})$ that increases $\mathbf{g}$. By the definition of a transition, applying some transition (r, factor) decreases $\boldsymbol{\kappa}[r.\mathit{from}]$ by factor. As by assumption on TAs, r is not in a cycle, $\boldsymbol{\kappa}[r.\mathit{from}]$ is increased only finitely often, namely, at most $N(\mathbf{p})$ times. As there are only finitely many rules in a TA, the proposition follows. □

3. Verification Problems: Parameterized Reachability vs. Safety & Liveness.

In this section we will discuss the verification problems for fault-tolerant distributed algorithms. A central challenge is to handle resilience conditions precisely.

Example 3.1. The safety property (unforgeability) of [64] expressed in terms of Figure 2 means that no process should ever enter ℓ_3 if initially all processes are in ℓ_0, given that $n > 3t \wedge t \ge f \ge 0$. We can express this in the counter system: under the resilience condition $n > 3t \wedge t \ge f \ge 0$, given an initial configuration σ, with $\sigma.\boldsymbol{\kappa}[\ell_0] = n - f$, to verify safety, we have to establish the absence of a schedule τ that satisfies $\sigma' = \tau(\sigma)$ and $\sigma'.\boldsymbol{\kappa}[\ell_3] > 0$.

In order to be able to answer this question, we have to deal with these resilience conditions precisely: Observe that ℓ_3 is unreachable, as all outgoing transitions from ℓ_0 contain guards that evaluate to false initially, and since all processes are in ℓ_0 no process ever increases x. A slight modification of $t \ge f$ to $t + 1 \ge f$ in the resilience condition changes the result, i.e., one fault too many breaks the system. For example, if $n = 4$, $t = 1$, and $f = 2$, then the new resilience condition holds, but as the guard $\gamma_1 : x \ge (t+1) - f$ is now initially true, then one correct process can fire the rule r_2 and increase x. Now when $x = 1$, the guard $\gamma_2 : x \ge (n - t) - f$ becomes true, so that the process can fire the rule r_4 and reach the state ℓ_3. This tells us that unforgeability is not satisfied in the system where the resilience condition is $n > 3t \wedge t + 1 \ge f \ge 0$. ◁

$$
\begin{aligned}
\psi &::= \mathit{pform} \mid \mathbf{G}\,\psi \mid \mathbf{F}\,\psi \mid \psi \wedge \psi \\
\mathit{pform} &::= \mathit{cform} \mid \mathit{gform} \vee \mathit{cform} \\
\mathit{cform} &::= \bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0 \mid \bigwedge_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] = 0 \mid \mathit{cform} \wedge \mathit{cform} \\
\mathit{gform} &::= \mathit{guard} \mid \neg \mathit{gform} \mid \mathit{gform} \wedge \mathit{gform}
\end{aligned}
$$

Table 1. The syntax of $\mathsf{ELTL_{FT}}$-formulas: *pform* defines propositional formulas, and ψ defines temporal formulas. We assume that $\mathit{Locs} \subseteq \mathcal{L}$ and $\mathit{guard} \in \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}$.

This is the verification question studied in [42], which can be formalized as follows:

Definition 3.2 (Parameterized reachability). *Given a threshold automaton TA and a Boolean formula B over $\{\boldsymbol{\kappa}[i] = 0 \mid i \in \mathcal{L}\}$, check whether there are parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$ with $\sigma_0.\mathbf{p} = \mathbf{p}$ and a finite schedule τ applicable to σ_0 such that $\tau(\sigma_0) \models B$.*

As shown in [42], if such a schedule exists, then there is also a schedule of bounded length. In this paper, we do not limit ourselves to reachability, but consider specifications of *counterexamples to safety and liveness* of FTDAs from the literature. We observe that such specifications use a simple subset of linear temporal logic that contains only the temporal operators $\mathbf{F}$ and $\mathbf{G}$.

Example 3.3. Consider a liveness property from the distributed algorithms literature called correctness [64]:

$$\mathbf{G}\,\mathbf{F}\,\psi_{\mathrm{fair}} \rightarrow (\boldsymbol{\kappa}[\ell_0] = 0 \rightarrow \mathbf{F}\,\boldsymbol{\kappa}[\ell_3] \neq 0). \quad (1)$$

Formula ψ_{fair} expresses the reliable communication assumption of distributed algorithms [32]. In this example, $\psi_{\mathrm{fair}} \equiv \boldsymbol{\kappa}[\ell_1] = 0 \wedge (x \ge t+1 \rightarrow \boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0) \wedge (x \ge n-t \rightarrow \boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_2] = 0)$. Intuitively, $\mathbf{G}\,\mathbf{F}\,\psi_{\mathrm{fair}}$ means that all processes in ℓ_1 should eventually leave this state, and if sufficiently many messages of type x are sent (γ_1 or γ_2 holds true), then all processes eventually receive them. If they do so, they have to eventually fire rules r_1, r_2, r_3, or r_4 and thus leave locations ℓ_0, ℓ_1, and ℓ_2. Our approach is based on possible shapes of *counterexamples*. Therefore, we consider the negation of the specification (1), that is, $\mathbf{G}\,\mathbf{F}\,\psi_{\mathrm{fair}} \wedge \boldsymbol{\kappa}[\ell_0] = 0 \wedge \mathbf{G}\,\boldsymbol{\kappa}[\ell_3] = 0$. In the following we define the logic that can express such counterexamples. ◁

The fragment of LTL limited to $\mathbf{F}$ and $\mathbf{G}$ was studied in [29, 46]. We further restrict it to the logic that we call *Fault-Tolerant Temporal Logic* ($\mathsf{ELTL_{FT}}$), whose syntax is shown in Table 1. The formulas derived from *cform* — called counter formulas — restrict counters, while the formulas derived from *gform* — called guard formulas — restrict shared variables. The formulas derived from *pform* are propositional formulas. The temporal operators $\mathbf{F}$ and $\mathbf{G}$ follow the standard semantics [5, 17], that is, for a configuration σ and an infinite schedule τ, it holds that $\mathsf{path}(\sigma, \tau) \models \varphi$, if:

1. $\sigma \models \varphi$, when φ is a propositional formula,
2. $\exists \tau', \tau'' : \tau = \tau' \cdot \tau''.\ \mathsf{path}(\tau'(\sigma), \tau'') \models \psi$, when $\varphi = \mathbf{F}\,\psi$,
3. $\forall \tau', \tau'' : \tau = \tau' \cdot \tau''.\ \mathsf{path}(\tau'(\sigma), \tau'') \models \psi$, when $\varphi = \mathbf{G}\,\psi$.

To stress that the formula should be satisfied by *at least one path*, we prepend $\mathsf{ELTL_{FT}}$-formulas with the existential path quantifier $\mathbf{E}$. We use the shorthand notation *true* for a valid propositional formula, e.g., $\bigwedge_{i \in \emptyset} \boldsymbol{\kappa}[i] = 0$. We also denote with $\mathsf{ELTL_{FT}}$ the set of all formulas that can be written using the logic $\mathsf{ELTL_{FT}}$.

We will reason about invariants of the finite subschedules, and consider a propositional formula ψ. Given a configuration σ, a finite schedule τ applicable to σ, and ψ, by $\mathsf{Cfgs}(\sigma, \tau) \models \psi$ we denote

```
algorithm parameterized_model_checking(TA, φ): // see Def. 3.4
 𝒢 := cut_graph (φ) /* Sect. 4 */
 ℋ := threshold_graph(TA) /* Sect. 5 */
 for each ≺ in topological_orderings(𝒢 ∪ ℋ) do // e.g., using [13]
  check_one_order(TA, φ, 𝒢, ℋ, ≺) /* Sect. 6–7 */
  if SMT_sat() then report the SMT model as a counterexample
```

Figure 3. A high-level description of the verification algorithm. For details of `check_one_order`, see Section 7.2 and Figure 10.

that ψ holds in every configuration σ' visited by the path $\mathsf{path}(\sigma, \tau)$. In other words, for every prefix τ' of τ, we have that $\tau'(\sigma) \models \psi$.

Definition 3.4 (Parameterized unsafety & non-liveness)**.** *Given a threshold automaton TA and an* $\mathsf{ELTL_{FT}}$ *formula* ψ*, check whether there are parameter values* $\mathbf{p} \in \mathbf{P}_{RC}$*, an initial configuration* $\sigma_0 \in I$ *with* $\sigma_0.\mathbf{p} = \mathbf{p}$*, and an infinite schedule* τ *of Sys(TA) applicable to* σ_0 *such that* $\mathsf{path}(\sigma_0, \tau) \models \psi$*.*

Complete bounded model checking. We solve this problem by showing how to reduce it to bounded model checking while guaranteeing completeness. To this end, we have to construct a bounded-length encoding of infinite schedules. In more detail:

- We observe that if $\mathsf{path}(\sigma_0, \tau) \models \psi$, then there is an initial state σ and two finite schedules ϑ and ρ (of unknown length) that can be used to construct an infinite (lasso-shaped) schedule $\vartheta \cdot \rho^\omega$, such that $\mathsf{path}(\sigma, \vartheta \cdot \rho^\omega) \models \psi$ (Section 4.1).
- Now given ϑ and ρ, we prove that we can use a ψ-specific reduction, to cut ϑ and ρ into subschedules $\vartheta_1, \dots, \vartheta_m$ and $\rho_1, \dots, \rho_n$, respectively so that the subschedules satisfy subformulas of ψ (Sections 4.2, 4.3 and 5).
- We use an offline partial order reduction, specific to the subformulas of ψ, and acceleration to construct representative schedules $\mathsf{rep}[\vartheta_i]$ and $\mathsf{rep}[\rho_j]$ that satisfy the required $\mathsf{ELTL_{FT}}$ formulas that are satisfied ϑ_i and ρ_j, respectively for $1 \le i \le m$ and $1 \le j \le n$. Moreover, $\mathsf{rep}[\vartheta_i]$ and $\mathsf{rep}[\rho_j]$ are fixed sequences of rules, where bounds on the lengths of the sequences are known (Section 6).
- These fixed sequence of rules can be used to encode a query to the SMT solver (Section 7.1). We ask whether there is an applicable schedule in the counter system that satisfies the sequence of rules and ψ (Section 7.3). If the SMT solver reports a contradiction, there exists no counterexample.

Based on these theoretical results, our tool implements the high-level verification algorithm from Figure 3 (in the comments we give the sections that are concerned with the respective step):

4. Shapes of Schedules that Satisfy $\mathsf{ELTL_{FT}}$

We characterize all possible shapes of lasso schedules that satisfy an $\mathsf{ELTL_{FT}}$-formula φ. These shapes are characterized by so-called *cut points*: We show that every lasso satisfying φ has a fixed number of cut points, one cut point per a subformula of φ that starts with $\mathbf{F}$. The configuration in the cut point of a subformula $\mathbf{F}\,\psi$ must satisfy ψ, and all configurations between two cut points must satisfy certain propositional formulas, which are extracted from the subformulas of φ that start with $\mathbf{G}$. Our notion of a cut point is motivated by extreme appearances of temporal operators [29].

Example 4.1. Consider the $\mathsf{ELTL_{FT}}$ formula $\varphi \equiv \mathbf{E}\,\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$, where $a, \dots, e$ are propositional formulas, whose structure is not of interest in this section. Formula φ is satisfiable by certain paths that have lasso shapes, i.e., a path consists of a finite prefix and a loop, which is repeated infinitely. These lassos may differ in the actual occurrences of the propositions and the start of the loop: For instance, at some point, a holds, and since then b always holds, then d holds at some point, then e holds at some point, then

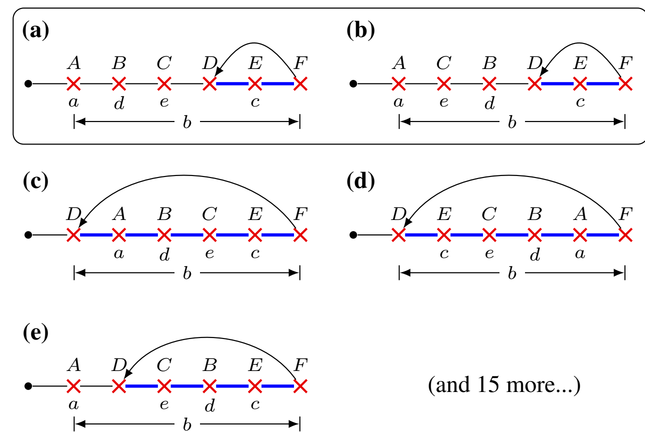


Figure 4. The shapes of lassos that satisfy the formula $\mathbf{E}\,\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$. The crosses show cut points for: (A) formula $\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$, (B) formula $\mathbf{F}\,d$, (C) formula $\mathbf{F}\,e$, (D) loop start, (E) formula $\mathbf{F}\,c$, and (F) loop end.

the loop is entered, and c holds infinitely often inside the loop. This is the case (a) shown in Figure 4, where the configurations in the cut points A, B, C, and D must satisfy the propositional formulas a, d, e, and c respectively, and the configurations between A and F must satisfy the propositional formula b. This example does not restrict the propositions between the initial state and the cut point A, so that this lasso shape, for instance, also captures the path where b holds from the beginning. There are 20 different lasso shapes for φ, five of them are shown in the figure. We construct lasso shapes that are sufficient for finding a path satisfying an $\mathsf{ELTL_{FT}}$ formula. In this example, it is sufficient to consider lasso shapes (a) and (b), since the other shapes can be constructed from (a) and (b) by unrolling the loop several times. ◁

4.1 Restricting Schedules to Lassos

In the seminal paper [66], Vardi and Wolper showed that if a finite-state transition system M *violates* an LTL formula — which requires *all paths* to satisfy the formula — then there is a path in M that (i) violates the formula and (ii) has lasso shape. As our logic $\mathsf{ELTL_{FT}}$ specifies counterexamples to the properties of fault-tolerant distributed algorithms, we are interested in this result in the following form: if the transition system *satisfies* an ELTL formula — which requires *one path* to satisfy the formula — then M has a path that (i) *satisfies* the formula and (ii) has lasso shape.

As observed above, counter systems are infinite state. Consequently, one cannot apply the results of [66] directly. However, using Proposition 2.10, we show that a similar result holds for counter systems of threshold automata and $\mathsf{ELTL_{FT}}$:

Proposition 4.2. *Given a threshold automaton TA and an* $\mathsf{ELTL_{FT}}$ *formula* φ*, if Sys(TA)* $\models \mathbf{E}\,\varphi$*, then there are an initial configuration* $\sigma_1 \in I$ *and a schedule* $\tau \cdot \rho^\omega$ *with the following properties:*

1. *the path satisfies the formula:* $\mathsf{path}(\sigma_1, \tau \cdot \rho^\omega) \models \varphi$*,*
2. *application of* ρ *forms a cycle:* $\rho^k(\tau(\sigma_1)) = \tau(\sigma_1)$ *for* $k \ge 0$*.*

Although in [43] we use Büchi automata to prove Proposition 4.2, we do not use Büchi automata in this paper. Since $\mathsf{ELTL_{FT}}$ uses only the temporal operators $\mathbf{F}$ and $\mathbf{G}$, we found it much easier to reason about the structure of $\mathsf{ELTL_{FT}}$ formulas directly (in the spirit of [29]) and then apply path reductions, rather than constructing the synchronous product of a Büchi automaton and of a counter system and then finding proper path reductions.

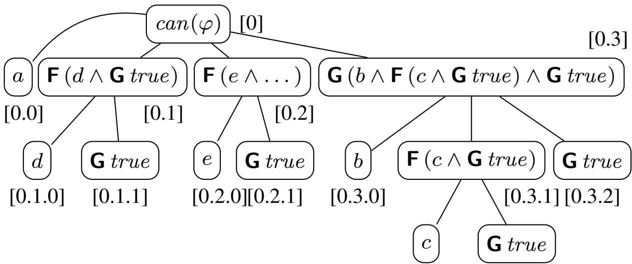


Figure 5. A canonical syntax tree of the $\mathsf{ELTL_{FT}}$ formula $\varphi \equiv \mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$ considered in Example 4.1. The labels $[w]$ denote identifiers of the tree nodes.

Although Proposition 4.2 guarantees counterexamples of lasso shape, it is not sufficient for model checking: (i) counter systems are infinite state, so that state enumeration may not terminate, and (ii) Proposition 4.2 does not provide us with bounds on the length of the lassos needed for bounded model checking. In the next section, we show how to split a lasso schedule in finite segments and to find constraints on lasso schedules that satisfy an $\mathsf{ELTL_{FT}}$ formula. In Section 6 we then construct shorter (bounded length) segments.

4.2 Characterizing Shapes of Lasso Schedules

We now construct a cut graph of an $\mathsf{ELTL_{FT}}$ formula: Cut graphs constrain the orders in which subformulas that start with the operator $\mathbf{F}$ are witnessed by configurations. The nodes of a cut graph correspond to cut points, while the edges constrain the order between the cut points. Using cut points, we give necessary and sufficient conditions for a lasso to satisfy an $\mathsf{ELTL_{FT}}$ formula in Theorems 4.12 and 4.13. Before defining cut graphs, we give the technical definitions of canonical formulas and canonical syntax trees.

Definition 4.3. *We inductively define canonical $\mathsf{ELTL_{FT}}$ formulas:*

- *if p is a propositional formula, then the formula $p \wedge \mathbf{G}\,true$ is a canonical formula of rank 0,*
- *if p is a propositional formula and formulas $\psi_1, \dots, \psi_k$ are canonical formulas (of any rank) for some $k \geq 1$, then the formula $p \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\,true$ is a canonical formula of rank 1,*
- *if p is a propositional formula and formulas $\psi_1, \dots, \psi_k$ are canonical formulas (of any rank) for some $k \geq 0$, and ψ_{k+1} is a canonical formula of rank 0 or 1, then the formula $p \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\,\psi_{k+1}$ is a canonical formula of rank 2.*

Example 4.4. Let p and q be propositional formulas. The formulas $p \wedge \mathbf{G}\,true$ and $true \wedge \mathbf{F}\,(q \wedge \mathbf{G}\,true) \wedge \mathbf{G}\,(p \wedge \mathbf{G}\,true)$ are canonical, while the formulas p, $\mathbf{F}\,q$, and $\mathbf{G}\,p$ are not canonical. Continuing Example 4.1, the canonical version of the formula $\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$ is the formula $\mathbf{F}\,(a \wedge \mathbf{F}\,(d \wedge \mathbf{G}\,true) \wedge \mathbf{F}\,(e \wedge \mathbf{G}\,true) \wedge \mathbf{G}\,(b \wedge \mathbf{F}\,(c \wedge \mathbf{G}\,true) \wedge \mathbf{G}\,true))$. ◁

We will use formulas in the following canonical form in order to simplify presentation.

Observation 1. *The properties of canonical $\mathsf{ELTL_{FT}}$ formulas:*

1. *Every canonical formula consists of canonical subformulas of the form $p \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\,\psi_{k+1}$ for some $k \geq 0$, for a propositional formula p, canonical formulas $\psi_1, \dots, \psi_k$, and a formula ψ_{k+1} that is either canonical, or equals to true.*
2. *If a canonical formula contains a subformula $\mathbf{G}\,(\dots \wedge \mathbf{G}\psi)$, then ψ equals true.*

Proposition 4.5. *There is a function $can : \mathsf{ELTL_{FT}} \to \mathsf{ELTL_{FT}}$ that produces for each formula $\varphi \in \mathsf{ELTL_{FT}}$ an equivalent canonical formula $can(\varphi)$.*

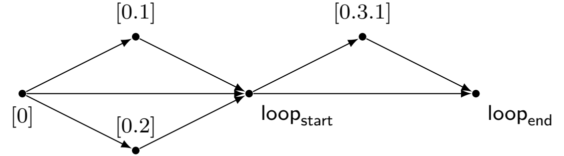


Figure 6. The cut graph of the canonical syntax tree in Figure 5

For an $\mathsf{ELTL_{FT}}$ formula, there may be several equivalent canonical formulas, e.g., $p \wedge \mathbf{F}\,(q \wedge \mathbf{G}\,true) \wedge \mathbf{F}\,(p \wedge \mathbf{G}\,true) \wedge \mathbf{G}\,true$ and $p \wedge \mathbf{F}\,(p \wedge \mathbf{G}\,true) \wedge \mathbf{F}\,(q \wedge \mathbf{G}\,true) \wedge \mathbf{G}\,true$ differ in the order of $\mathbf{F}$-subformulas. With the function can we fix one such a formula.

Canonical syntax trees. The canonical syntax tree of the formula introduced in Example 4.1 is shown in Figure 5. With $\mathbb{N}_0^*$ we denote the set of all finite words over natural numbers — these words are used as node identifiers.

Definition 4.6. *The* canonical syntax tree *of a formula $\varphi \in \mathsf{ELTL_{FT}}$ is the set $\mathcal{T}(\varphi) \subseteq \mathsf{ELTL_{FT}} \times \mathbb{N}_0^*$ constructed inductively as follows:*

1. *The tree contains the root node labeled with the canonical formula $can(\varphi)$ and id 0, that is, $\langle can(\varphi), 0\rangle \in \mathcal{T}(\varphi)$.*
2. *Consider a tree node $\langle \psi, w\rangle \in \mathcal{T}(\varphi)$ such that for some canonical formula $\psi' \in \mathsf{ELTL_{FT}}$ one of the following holds: (a) $\psi = \psi' = can(\varphi)$, or (b) $\psi = \mathbf{F}\psi'$, or (c) $\psi = \mathbf{G}\psi'$.*
   If ψ' is $p \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\,\psi_{k+1}$ for some $k \geq 0$, then the tree $\mathcal{T}(\varphi)$ contains a child node for each of the conjuncts of ψ', that is, $\langle p, w.0\rangle \in \mathcal{T}(\varphi)$, as well as $\langle \mathbf{F}\psi_i, w.i\rangle \in \mathcal{T}(\varphi)$ and $\langle \mathbf{G}\psi_j, w.j\rangle \in \mathcal{T}(\varphi)$ for $1 \leq i \leq k$ and $j = k+1$.

Observation 2. *The canonical syntax tree $\mathcal{T}(\varphi)$ of an $\mathsf{ELTL_{FT}}$ formula φ has the following properties:*

- *Every node $\langle \psi, w\rangle$ has the unique identifier w, which encodes the path to the node from the root.*
- *Every intermediate node is labeled with a temporal operator $\mathbf{F}$ or $\mathbf{G}$ over the conjunction of the formulas in the children nodes.*
- *The root node is labeled with the formula φ itself, and φ is equivalent to the conjunction of the root's children formulas, possibly preceded with a temporal operator $\mathbf{F}$ or $\mathbf{G}$.*

The temporal formulas that appear under the operator $\mathbf{G}$ have to be dealt with by the loop part of a lasso. To formalize this, we say that a node with id $w \in \mathbb{N}_0^*$ is *covered* by a $\mathbf{G}$-node, if w can be split into two words $u_1, u_2 \in \mathbb{N}_0^*$ with $w = u_1.u_2$, and there is a formula $\psi \in \mathsf{ELTL_{FT}}$ such that $\langle \mathbf{G}\,\psi, u_1\rangle \in \mathcal{T}(\varphi)$.

Cut graphs. Using the canonical syntax tree $\mathcal{T}(\varphi)$ of a formula φ, we capture in a so-called *cut graph* the possible orders in which formulas $\mathbf{F}\,\psi$ should be witnessed by configurations of a lasso-shaped path. We will then use the occurrences of the formula ψ to cut the lasso into bounded finite schedules.

Example 4.7. Figure 6 shows the cut graph of the canonical syntax tree in Figure 5. It consists of tree node ids for subformulas starting with $\mathbf{F}$, and two special nodes for the start and the end of the loop. In the cut graph, the node with id 0 precedes the node with id 0.1, since at least one configuration satisfying $(a \wedge \mathbf{F}\,(d \wedge \dots) \wedge \dots)$ should occur on a path before (or at the same moment as) a state satisfying $(d \wedge \dots)$. Similarly, the node with id 0 precedes the node with id 0.2. The nodes with ids 0.1 and 0.2 do not have to precede each other, as the formulas d and e can be satisfied in either order. Since the nodes with the ids 0, 0.1, and 0.2 are not covered by a $\mathbf{G}$-node, they both precede the loop start. The loop start precedes the node with id 0.3.1, as this node is covered by a $\mathbf{G}$-node. ◁

Definition 4.8. *The* cut graph *$\mathcal{G}(\varphi)$ of an $\mathsf{ELTL_{FT}}$ formula is a directed acyclic graph $(\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ with the following properties:*

1. *The set of nodes* $\mathcal{V}_\mathcal{G} = \{\mathsf{loop}_{\mathsf{start}}, \mathsf{loop}_{\mathsf{end}}\} \cup \{w \in \mathbb{N}_0^* \mid \exists \psi.\ \langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)\}$ *contains the tree ids that label* $\mathbf{F}$*-formulas and two special nodes* $\mathsf{loop}_{\mathsf{start}}$ *and* $\mathsf{loop}_{\mathsf{end}}$*, which denote the start and the end of the loop respectively.*
2. *The set of edges* $\mathcal{E}_\mathcal{G}$ *satisfies the following constraints:*
   (a) *Each tree node* $\langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)$ *that is* not *covered by a* $\mathbf{G}$*-node precedes the loop start, i.e.,* $(w, \mathsf{loop}_{\mathsf{start}}) \in \mathcal{E}_\mathcal{G}$*.*
   (b) *For each tree node* $\langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)$ *covered by a* $\mathbf{G}$*-node:*
   - *the loop start precedes* w*, i.e.,* $(\mathsf{loop}_{\mathsf{start}}, w) \in \mathcal{E}_\mathcal{G}$*, and*
   - w *precedes the loop end, i.e.,* $(w, \mathsf{loop}_{\mathsf{end}}) \in \mathcal{E}_\mathcal{G}$*.*

   (c) *For each pair of tree nodes* $\langle \mathbf{F}\psi_1, w\rangle, \langle \mathbf{F}\psi_2, w.i\rangle \in \mathcal{T}(\varphi)$ *not covered by a* $\mathbf{G}$*-node, we require* $(w, w.i) \in \mathcal{E}_\mathcal{G}$*.*
   (d) *For each pair of tree nodes* $\langle \mathbf{F}\psi_1, w_1\rangle, \langle \mathbf{F}\psi_2, w_2\rangle \in \mathcal{T}(\varphi)$ *that are both covered by a* $\mathbf{G}$*-node, we require either* $(w_1, w_2) \in \mathcal{E}_\mathcal{G}$*, or* $(w_2, w_1) \in \mathcal{E}_\mathcal{G}$ *(but not both).*

Definition 4.9. *Given a lasso* $\tau \cdot \rho^\omega$ *and a cut graph* $\mathcal{G}(\varphi) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$*, we call a function* $\zeta : \mathcal{V}_\mathcal{G} \to \{0, \dots, |\tau| + |\rho| - 1\}$ *a* cut function*, if the following holds:*
- $\zeta(\mathsf{loop}_{\mathsf{start}}) = |\tau|$ *and* $\zeta(\mathsf{loop}_{\mathsf{end}}) = |\tau| + |\rho| - 1$*,*
- *if* $(v, v') \in \mathcal{E}_\mathcal{G}$*, then* $\zeta(v) \le \zeta(v')$*.*

We call the indices $\{\zeta(v) \mid v \in \mathcal{V}_\mathcal{G}\}$ the *cut points*. Given a schedule τ and an index $k : 0 \le k < |\tau| + |\rho|$, we say that the index k *cuts* τ into π' and π'', if $\tau = \pi' \cdot \pi''$ and $|\pi'| = k$.

Informally, for a tree node $\langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)$, a cut point $\zeta(w)$ witnesses satisfaction of $\mathbf{F}\psi$, that is, the formula ψ holds at the configuration located at the cut point. It might seem that Definitions 4.8 and 4.9 are too restrictive. For instance, assume that the node $\langle \mathbf{F}\psi, w\rangle$ is not covered by a $\mathbf{G}$-node, and there is a lasso schedule $\tau \cdot \rho^\omega$ that satisfies the formula φ at a configuration σ. It is possible that the formula ψ is witnessed only by a cut point inside the loop. At the same time, Definition 4.9 forces $\zeta(w) \le \zeta(\mathsf{loop}_{\mathsf{start}})$. We show that this problem is resolved by unwinding the loop K times for some $K \ge 0$, so that there is a cut function for the lasso with the prefix $\tau \cdot \rho^K$ and the loop ρ:

Proposition 4.10. *Let* φ *be an* $\mathsf{ELTL}_{\mathsf{FT}}$ *formula,* σ *be a configuration and* $\tau \cdot \rho^\omega$ *be a lasso schedule applicable to* σ *such that* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ *holds. There is a constant* $K \ge 0$ *and a cut function* ζ *such that for every* $\langle \mathbf{F}\psi, w\rangle \in \mathcal{G}(\mathcal{T}(\varphi))$ *if* $\zeta(w)$ *cuts* $(\tau \cdot \rho^K) \cdot \rho$ *into* π' *and* π''*, then* ψ *is satisfied at the cut point, that is,* $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$*.*

Proof sketch. The detailed proof is given in [43]. We will present the required constant $K \ge 0$ and the cut function ζ. To this end, we use extreme appearances of $\mathbf{F}$-formulas (cf. [29, Sec. 4.3]) and use them to find ζ. An extreme appearance of a formula $\mathbf{F}\psi$ is the furthest point in the lasso that still witnesses ψ. There might be a subformula that is required to be witnessed in the prefix, but in $\tau \cdot \rho^\omega$ it is only witnessed by the loop. To resolve this, we replace τ by a a longer prefix $\tau \cdot \rho^K$, by unrolling the loop ρ several times; more precisely, K times, where K is the number of nodes that should precede the lasso start. In other words, if all extreme appearances of the nodes happen to be in the loop part, and they appear in the order that is against the topological order of the graph $\mathcal{G}(\mathcal{T}(\varphi))$, we unroll the loop K times (the number of nodes that have to be in the prefix) to find the prefix, in which the nodes respect the topological order of the graph. In the unrolled schedule we can now find extreme appearances of the required subformulas in the prefix. □

We show that to satisfy an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, a lasso should (i) satisfy propositional subformulas of $\mathbf{F}$-formulas in the respective cut points, and (ii) maintain the propositional formulas of $\mathbf{G}$-formulas from some cut point on. This is formalized as a witness.

In the following definition, we use a short-hand notation for propositional subformulas: given an $\mathsf{ELTL}_{\mathsf{FT}}$-formula ψ and its canonical form $can(\psi) = \psi_0 \land \mathbf{F}\psi_1 \land \dots \land \mathbf{F}\psi_k \land \mathbf{G}\psi_{k+1}$, we use the notation $prop(\psi)$ to denote the formula ψ_0.

Definition 4.11. *Given a configuration* σ*, a lasso* $\tau \cdot \rho^\omega$ *applicable to* σ*, and an* $\mathsf{ELTL}_{\mathsf{FT}}$ *formula* φ*, a cut function* ζ *of* $\mathcal{G}(\mathcal{T}(\varphi))$ *is a* witness *of* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$*, if the three conditions hold:*

(C1) *For* $can(\varphi) \equiv \psi_0 \land \mathbf{F}\psi_1 \land \dots \land \mathbf{F}\psi_k \land \mathbf{G}\psi_{k+1}$*:*
(a) $\sigma \models \psi_0$*, and*
(b) $\mathsf{Cfgs}(\sigma, \tau \cdot \rho) \models prop(\psi_{k+1})$*.*

(C2) *For* $\langle \mathbf{F}\psi, v\rangle \in \mathcal{T}(\varphi)$ *with* $\zeta(v) < |\tau|$*, if* $\zeta(v)$ *cuts* $\tau \cdot \rho$ *into* π' *and* π'' *and* $\psi \equiv \psi_0 \land \mathbf{F}\psi_1 \land \dots \land \mathbf{F}\psi_k \land \mathbf{G}\psi_{k+1}$*, then:*
(a) $\pi'(\sigma) \models \psi_0$*, and*
(b) $\mathsf{Cfgs}(\pi'(\sigma), \pi'') \models prop(\psi_{k+1})$*.*

(C3) *For* $\langle \mathbf{F}\psi, v\rangle \in \mathcal{T}(\varphi)$ *with* $\zeta(v) \ge |\tau|$*, if* $\zeta(v)$ *cuts* $\tau \cdot \rho$ *into* π' *and* π'' *and* $\psi \equiv \psi_0 \land \mathbf{F}\psi_1 \land \dots \land \mathbf{F}\psi_k \land \mathbf{G}\psi_{k+1}$*, then:*
(a) $\pi'(\sigma) \models \psi_0$*, and*
(b) $\mathsf{Cfgs}(\tau(\sigma), \rho) \models prop(\psi_{k+1})$*.*

Conditions (a) require that propositional formulas hold in a configuration, while conditions (b) require that propositional formulas hold on a finite suffix. Hence, to ensure that a cut function constitutes a witness, one has to check the configurations of a *fixed number of finite* paths (between the cut points). This property is crucial for the path reduction (see Section 6). Theorems 4.12 and 4.13 show that the existence of a witness is a sound and complete criterion for the existence of a lasso satisfying an $\mathsf{ELTL}_{\mathsf{FT}}$ formula.

Theorem 4.12 (Soundness)**.** *Let* σ *be a configuration,* $\tau \cdot \rho^\omega$ *be a lasso applicable to* σ*, and* φ *be an* $\mathsf{ELTL}_{\mathsf{FT}}$ *formula. If there is a witness of* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$*, then the lasso* $\tau \cdot \rho^\omega$ *satisfies* φ*, that is* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$*.*

Theorem 4.13 (Completeness)**.** *Let* φ *be an* $\mathsf{ELTL}_{\mathsf{FT}}$ *formula,* σ *be a configuration and* $\tau \cdot \rho^\omega$ *be a lasso applicable to* σ *such that* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ *holds. There is a witness of* $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega) \models \varphi$ *for some* $K \ge 0$*.*

Theorem 4.12 is proven for subformulas of φ by structural induction on the intermediate nodes of the canonical syntax tree. In the proof of Theorem 4.13 we use Proposition 4.10 to prove the points of Definition 4.11. (The detailed proofs are given in [43].)

4.3 Using Cut Graphs to Enumerate Shapes of Lassos

Proposition 4.2 and Theorem 4.13 suggest that in order to find a schedule that satisfies an $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ, it is sufficient to look for lasso schedules that can be cut in such a way that the configurations at the cut points and the configurations between the cut points satisfy certain propositional formulas. In fact, the cut points as defined by cut functions (Definition 4.9) are *topological orderings* of the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$. Consequently, by enumerating the topological orderings of the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$ we can enumerate the *lasso shapes*, among which there is a lasso schedule satisfying φ (if φ holds on the counter system). These shapes differ in the order, in which $\mathbf{F}$-subformulas of φ are witnessed. For this, one can use fast generation algorithms, e.g., [13].

Example 4.14. Consider the cut graph in Figure 6. The ordering of its vertices $0, 0.1, 0.2, \mathsf{loop}_{\mathsf{start}}, 0.3.1, \mathsf{loop}_{\mathsf{end}}$ corresponds to the lasso shape (a) shown in Figure 4, while the ordering $\mathsf{loop}_{\mathsf{start}}, 0, 0.2, 0.1, \mathsf{loop}_{\mathsf{start}}, 0.3.1, \mathsf{loop}_{\mathsf{end}}$ corresponds to the lasso shape (b). These are the two lasso shapes that one has to analyze, and they are the result of our construction using the cut graph. The other 18 lasso shapes in the figure are not required, and not constructed by our method. ◁

From this observation, we conclude that given a topological ordering $v_1, \dots, v_{|\mathcal{V}_\mathcal{G}|}$ of the cut graph $\mathcal{G}(\mathcal{T}(\varphi)) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, one has to look for a lasso schedule that can be written as an alternating

sequence of configurations σ_i and schedules τ_j:

$$\sigma_0, \tau_0, \sigma_1, \tau_1, \dots, \sigma_\ell, \tau_\ell, \dots, \sigma_{|\mathcal{V}_\mathcal{G}|-1}, \tau_{|\mathcal{V}_\mathcal{G}|}, \sigma_{|\mathcal{V}_\mathcal{G}|}, \quad (2)$$

where $v_\ell = \mathsf{loop}_{\mathsf{start}}$, $v_{|\mathcal{V}_\mathcal{G}|} = \mathsf{loop}_{\mathsf{end}}$, and $\sigma_\ell = \sigma_{|\mathcal{V}_\mathcal{G}|}$. Moreover, by Definition 4.11, the sequence of configurations and schedules should satisfy ***(C1)–(C3)***, e.g., if a node v_i corresponds to the formula $\mathbf{F}\,(\psi_0 \wedge \dots \wedge \mathbf{G}\,\psi_{k+1})$ and this formula matches Condition ***(C2)***, then the following should hold:

1. Configuration σ_i satisfies the propositional formula: $\sigma_i \models \psi_0$.
2. All configurations visited by the schedule $\tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}$ from the configuration σ_i satisfy the propositional formula $prop(\psi_{k+1})$. Formally, $\mathsf{Cfgs}(\sigma_i, \tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}) \models prop(\psi_{k+1})$.

One can write an SMT query for the sequence (2) satisfying Conditions ***(C1)–(C3)***. However, this approach has two problems:

1. The order of rules in schedules $\tau_0, \dots, \tau_{|\mathcal{V}_\mathcal{G}|}$ is not fixed. Nondeterministic choice of rules complicates the SMT query.
2. To guarantee completeness of the search, one requires a bound on the length of schedules $\tau_0, \dots, \tau_{|\mathcal{V}_\mathcal{G}|}$.

For reachability properties these issues were addressed in [42] by showing that one only has to consider specific orders of the rules; so-called representative schedules. To lift this technique to $\mathsf{ELTL_{FT}}$, we are left with two issues:

1. The shortening technique applies to steady schedules, i.e., the schedules that do not change evaluation of the guards. Thus, we have to break the schedules $\tau_0, \dots, \tau_{|\mathcal{V}_\mathcal{G}|}$ into steady schedules. This issue is addressed in Section 5.
2. The shortening technique preserves state reachability, e.g., after shortening of τ_i, the resulting schedule still reaches configuration σ_{i+1}. But it may violate an invariant such as $\mathsf{Cfgs}(\sigma_i, \tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}) \models prop(\psi_{k+1})$. This issue is addressed in Section 6.

5. Cutting Lassos with Threshold Guards

We introduce threshold graphs to cut a lasso into steady schedules, in order to apply the shortening technique of Section 6. Then, we combine the cut graphs and threshold graphs to cut a lasso into smaller finite segments, which can be first shortened and then checked with the approach introduced in Section 4.3.

Given a configuration σ, its context $\omega(\sigma)$ is the set that consists of the lower guards unlocked in σ and the upper guards locked in σ, i.e., $\omega(\sigma) = \Omega^{\mathrm{rise}} \cup \Omega^{\mathrm{fall}}$, where $\Omega^{\mathrm{rise}} = \{g \in \Phi^{\mathrm{rise}} \mid \sigma \models g\}$ and $\Omega^{\mathrm{fall}} = \{g \in \Phi^{\mathrm{fall}} \mid \sigma \not\models g\}$. As discussed in Example 2.9 on page 4, since the shared variables are never decreased, the contexts in a path are monotonically non-decreasing:

Proposition 5.1 (Prop. 3 of [42]). *If a transition t is enabled in a configuration σ, then $\omega(\sigma) \subseteq \omega(t(\sigma))$.*

Example 5.2. Continuing Example 2.9, which considers the TA in Figure 2. Both threshold guards γ_1 and γ_2 are false in the initial state σ. Thus, $\omega(\sigma) = \emptyset$. The transition $t = (r_1, 1)$ unlocks the guard γ_1, i.e., $\omega(t(\sigma)) = \{\gamma_1\}$. ◁

As the transitions of the counter system $\mathsf{Sys}(\mathsf{TA})$ never decrease shared variables, the loop of a lasso schedule must be steady:

Proposition 5.3. *For each configuration σ and a schedule $\tau \cdot \rho^\omega$, if $\rho^k(\tau(\sigma)) = \tau(\sigma)$ for $k \geq 0$, then the loop ρ is steady for $\tau(\sigma)$, that is, $\omega(\rho(\tau(\sigma))) = \omega(\tau(\sigma))$.*

In [42], Proposition 5.1 was used to cut a finite path into segments, one per context. We introduce threshold graphs and their topological orderings to apply this idea to lasso schedules.

Definition 5.4. *A* threshold graph *is $\mathcal{H}(TA) = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ such that:*

- *The vertices set $\mathcal{V}_\mathcal{H}$ contains the threshold guards and the special node $\mathsf{loop}_{\mathsf{start}}$, i.e., $\mathcal{V}_\mathcal{H} = \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}} \cup \{\mathsf{loop}_{\mathsf{start}}\}$.*

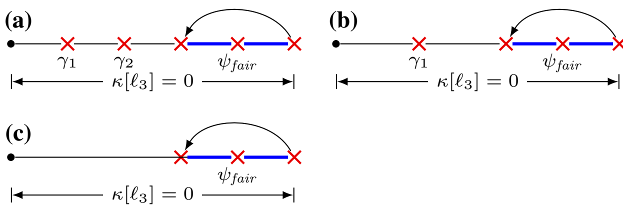


Figure 7. The shapes of lassos to check the correctness property in Example 3.3. Recall that γ_1 and γ_2 are the threshold guards, defined as $x \geq t + 1 - f$ and $x \geq n - t - f$ respectively.

- *There is an edge from a guard $g_1 \in \Phi^{\mathrm{rise}}$ to a guard $g_2 \in \Phi^{\mathrm{rise}}$, if g_2 cannot be unlocked before g_1, i.e., $(g_1, g_2) \in \mathcal{E}_\mathcal{H}$, if for each configuration $\sigma \in \Sigma$, $\sigma \models g_2$ implies $\sigma \models g_1$.*
- *There is an edge from a guard $g_1 \in \Phi^{\mathrm{fall}}$ to a guard $g_2 \in \Phi^{\mathrm{fall}}$, if g_2 cannot be locked before g_1, i.e., $(g_1, g_2) \in \mathcal{E}_\mathcal{H}$, if for each configuration $\sigma \in \Sigma$, $\sigma \not\models g_2$ implies $\sigma \not\models g_1$.*

Note that the conditions in Definition 5.4 can be easily checked with an SMT solver, for all configurations.

Example 5.5. The threshold graph of the TA in Figure 2 has the vertices $\mathcal{V}_\mathcal{H} = \{\gamma_1, \gamma_2, \mathsf{loop}_{\mathsf{start}}\}$ and the edges $\mathcal{E}_\mathcal{H} = \{(\gamma_1, \gamma_2)\}$. ◁

Similar to Section 4.3, we consider a topological ordering $g_1, \dots, g_\ell, \dots, g_{|\mathcal{V}_\mathcal{H}|}$ of the vertices of the threshold graph. The node $g_\ell = \mathsf{loop}_{\mathsf{start}}$ indicates the point where a loop should start, and thus by Proposition 5.3, after that point the context does not change. Thus, we consider only the subsequence $g_1, \dots, g_{\ell-1}$ and split the path $\mathsf{path}(\sigma, \tau \cdot \rho)$ of a lasso schedule $\tau \cdot \rho^\omega$ into an alternating sequence of configurations σ_i and schedules τ_0 and $t_j \cdot \tau_j$, for $1 \leq j < \ell$, ending up with the loop ρ (starting in $\sigma_{\ell-1}$ and ending in $\sigma_\ell = \sigma_{\ell-1}$):

$$\sigma_0, \tau_0, \sigma_1, (t_1 \cdot \tau_1), \dots, \sigma_{\ell-2}, (t_{\ell-1} \cdot \tau_{\ell-1}), \sigma_{\ell-1}, \rho, \sigma_\ell \quad (3)$$

In this sequence, the transitions $t_1, \dots, t_{\ell-1}$ change the context, and the schedules $\tau_0, \tau_1, \dots, \tau_{\ell-1}, \rho$ are steady. Finally, we interleave a topological ordering of the vertices of the cut graph with a topological ordering of the vertices of the threshold graph. More precisely, we use a topological ordering of the vertices of the union of the cut graph and the threshold graph. We use the resulting sequence to cut a lasso schedule following the approach in Section 4.3 (cf. Equation (2)). By enumerating all such interleavings, we obtain all lasso shapes. Again, the lasso is a sequence of steady schedules and context-changing transitions.

Example 5.6. Continuing Example 1 given on page 5, we consider the lasso shapes that satisfy the $\mathsf{ELTL_{FT}}$ formula $\mathbf{G}\,\mathbf{F}\,\psi_{\mathrm{fair}} \wedge \boldsymbol{\kappa}[\ell_0] = 0 \wedge \mathbf{G}\,\boldsymbol{\kappa}[\ell_3] = 0$. Figure 7 shows the lasso shapes that have to be inspected by an SMT solver. In case (a), both threshold guards γ_1 and γ_2 are eventually changed to true, while the counter $\kappa[\ell_3]$ is never increased in a fair execution. For $n = 3t$, this is actually a counterexample to the correctness property explained in Example 1. In cases (b) and (c) at most one threshold guard is eventually changed to true, so these lasso shapes cannot produce a counterexample. ◁

In the following section, we will show how to shorten steady schedules, while maintaining Conditions ***(C1)–(C3)*** of Definition 4.11, required to satisfy the $\mathsf{ELTL_{FT}}$ formula.

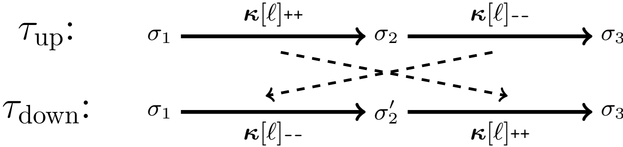


Figure 8. Changing the order of transitions can violate $\mathsf{ELTL}_{\mathsf{FT}}$ formulas. If $\sigma_1.\boldsymbol{\kappa}[\ell] = 1$, then for the upper schedule τ_{up} holds that $\mathsf{Cfgs}(\sigma_1, \tau_{\text{up}}) \models \boldsymbol{\kappa}[\ell] > 0$, while for the lower one this is not the case, because $\sigma_2' \not\models \boldsymbol{\kappa}[\ell] > 0$.

6. The Short Counterexample Property

Our verification approach focuses on counterexamples, and as discussed in Section 3, negations of specifications are expressed in $\mathsf{ELTL}_{\mathsf{FT}}$. In the case of reachability properties, counterexamples are finite schedules reaching a bad state from an initial state. An efficient method for finding counterexamples to reachability can be found in [42]. It is based on the short counterexample property. Namely, it was proven that for each threshold automaton, there is a constant d such that if there is a schedule that reaches a bad state, then there must also exist an accelerated schedule that reaches that state in at most d transitions (i.e., d is the diameter of the counter system). The proof in [42] is based on the following three steps:

1. each finite schedule (which may or may not be a counterexample), can be divided into a few steady schedules,
2. for each of these steady schedules they find a representative, i.e., an accelerated schedule of bounded length, with the same starting and ending configurations as the original schedule,
3. at the end, all these representatives are concatenated in the same order as the original steady schedules.

This result guarantees that the system is correct if no counterexample to reachability properties is found using bounded model checking with bound d. In this section, we extend the technique from Point 2 from reachability properties to $\mathsf{ELTL}_{\mathsf{FT}}$ formulas. The central result regarding Point 2 is the following proposition which is a specialization of [42, Prop. 7]:

Proposition 6.1. *Let $TA = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ be a threshold automaton. For every configuration σ and every steady schedule τ applicable to σ, there exists a steady schedule $\mathsf{srep}[\sigma, \tau]$ with the following properties: $\mathsf{srep}[\sigma, \tau]$ is applicable to σ, $\mathsf{srep}[\sigma, \tau](\sigma) = \tau(\sigma)$, and $|\mathsf{srep}[\sigma, \tau]| \le 2 \cdot |\mathcal{R}|$.*

We observe that the proposition talks about the first configuration σ and the last one $\tau(\sigma)$, while it ignores intermediate configurations. However, for $\mathsf{ELTL}_{\mathsf{FT}}$ formulas, one has to consider all configurations in a schedule, and not just the first and the last one.

Example 6.2. Figure 8 shows the result of swapping transitions. The approaches by [50] and [42] are only concerned with the first and last configurations: they use the property that after swapping transitions, σ_3 is still reached from σ_1. The arguments used in [42, 50] do not care about the fact that the resulting path visits a different intermediate state (σ_2' instead of σ_2). However, if $\sigma_1.\boldsymbol{\kappa}[\ell] = 1$, then $\sigma_2.\boldsymbol{\kappa}[\ell] > 0$, while $\sigma_2'.\boldsymbol{\kappa}[\ell] = 0$. Hence, swapping transitions may change the evaluation of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas, e.g., $\mathbf{G}\,(\boldsymbol{\kappa}[\ell] > 0)$. $\triangleleft$

Another challenge in verification of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas is that counterexamples to liveness properties are infinite paths. As discussed in Section 4, we consider infinite paths of lasso shape $\vartheta \cdot \rho^{\omega}$. For a finite part of a schedule, $\vartheta \cdot \rho$, satisfying an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, we show the existence of a new schedule, $\vartheta' \cdot \rho'$, of bounded length satisfying the same formula as the original one. Regarding the shortening, our approach uses a similar idea as the one from [42]. We follow modified steps from reachability analysis:

1. We split $\vartheta \cdot \rho$ into several steady schedules, using cut points introduced in Sections 4 and 5. The cut points depend not only on threshold guards, but also on the $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ representing the negation of a specification we want to check. Given such a steady schedule τ, each configuration of τ satisfies a set of propositional subformulas of φ, which are covered by the operator $\mathbf{G}$ in φ.
2. For each of these steady schedules we find a representative, that is, an accelerated schedule of bounded length that satisfies the necessary propositional subformulas as in the original schedule (i.e., not just that starting and ending configurations coincide).
3. We concatenate the obtained representatives in the original order.

In [43], we present the mathematical details for obtaining these representative schedules, and prove different cases that taken together establish our following main theorem:

Theorem 6.3. *Let $TA = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ be a threshold automaton, and let $Locs \subseteq \mathcal{L}$ be a set of locations. Let σ be a configuration, let τ be a steady conventional schedule applicable to σ, and let ψ be one of the following formulas:*

$$\bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0, \quad or \quad \bigwedge_{\ell \in Locs} \boldsymbol{\kappa}[\ell] = 0.$$

If all configurations visited by τ from σ satisfy ψ, i.e., $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, then there is a steady representative schedule $\mathsf{repr}[\psi, \sigma, \tau]$ with the following properties:

a) The representative is applicable, and ends in the same final state: $\mathsf{repr}[\psi, \sigma, \tau]$ is applicable to σ, and $\mathsf{repr}[\psi, \sigma, \tau](\sigma) = \tau(\sigma)$,
b) The representative has bounded length: $|\mathsf{repr}[\psi, \sigma, \tau]| \le 6 \cdot |\mathcal{R}|$,
c) The representative maintains the formula ψ. In other words, $\mathsf{Cfgs}(\sigma, \mathsf{repr}[\psi, \sigma, \tau]) \models \psi$,
d) The representative is a concatenation of three representative schedules srep from Proposition 6.1: there exist τ_1, τ_2 and τ_3, (possibly empty) subschedules of τ, such that $\tau_1 \cdot \tau_2 \cdot \tau_3$ is applicable to σ, and it holds that $(\tau_1 \cdot \tau_2 \cdot \tau_3)(\sigma) = \tau(\sigma)$, and $\mathsf{repr}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \mathsf{srep}[(\tau_1 \cdot \tau_2)(\sigma), \tau_3]$.

Our approach is slightly different in the case when the formula ψ has a more complex form: $\bigwedge_{1 \le m \le n} \bigvee_{\ell \in Locs_m} \boldsymbol{\kappa}[\ell] \neq 0$, for $Locs_m \subseteq \mathcal{L}$, where $1 \le m \le n$ and $n \in \mathbb{N}$. In this case, our proof requires the schedule τ to have sufficiently large counter values. To ensure that there is an infinite schedule with sufficiently large counter values, we first prove that if a counterexample exists in a small system, there also exists one in a larger system, that is, we consider configurations where each counter is multiplied with a constant *finite multiplier* μ. For resilience conditions that do not correspond to parameterized systems (i.e., fix the system size to, e.g., $n = 4$) or pathological threshold automata, such multipliers may not exist. However, all our benchmarks have multipliers, and existence of multipliers can easily be checked using simple queries to SMT solvers in preprocessing. This additional restriction leads to slightly smaller bounds on the lengths of representative schedules:

Theorem 6.4. *Fix a threshold automaton $TA = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ that has a finite multiplier μ, and a configuration σ. For an $n \in \mathbb{N}$, fix sets of locations $Locs_m \subseteq \mathcal{L}$ for $1 \le m \le n$. If $\psi = \bigwedge_{1 \le m \le n} \bigvee_{\ell \in Locs_m} \boldsymbol{\kappa}[\ell] \neq 0$, then for every steady conventional schedule τ, applicable to σ, with $\mathsf{Cfgs}(\sigma, \tau) \models \psi$, there exists a schedule $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ with the following properties:*

a) The representative is applicable and ends in the same final state: $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ is a steady schedule applicable to $\mu\sigma$, and $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau](\mu\sigma) = \mu\tau(\mu\sigma)$,
b) The representative has bounded length: $|\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]| \le 4 \cdot |\mathcal{R}|$,
c) The representative maintains the formula ψ. In other words, $\mathsf{Cfgs}(\mu\sigma, \mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]) \models \psi$,

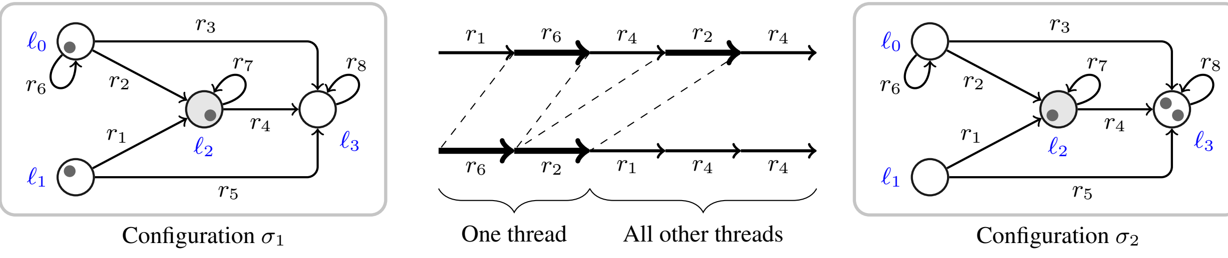


Figure 9. Example of constructing a representative schedule by moving a thread to the beginning. The number of dots in the local states correspond to counter values, i.e., $\sigma_1.\boldsymbol{\kappa}[\ell_0] = \sigma_1.\boldsymbol{\kappa}[\ell_1] = \sigma_1.\boldsymbol{\kappa}[\ell_2] = 1$ and $\sigma_1.\boldsymbol{\kappa}[\ell_3] = 0$.

d) The representative is a concatenation of two representative schedules srep *from Proposition 6.1:*
$\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau] = \mathsf{srep}[\mu\sigma, \tau] \cdot \mathsf{srep}[\tau(\mu\sigma), (\mu - 1)\tau]$.

The main technical challenge for proving Theorems 6.3 and 6.4 is that we want to swap transitions and maintain $\mathsf{ELTL}_{\mathsf{FT}}$ formulas at the same time. As discussed in Example 6.2, simply applying the ideas from the reachability analysis in [42, 50] is not sufficient.

We address this challenge by more refined swapping strategies depending on the property ψ of Theorem 6.3. For instance, the intuition behind $\bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$ is that in a given distributed algorithm, there should always be at least one process in one of the states in $Locs$. Hence, we would like to consider individual processes, but in the context of counter systems. Therefore, we introduce a mathematical notion we call a *thread*, which is a schedule that can be executed by an individual process. A thread is then characterized depending on whether it starts in $Locs$, ends in $Locs$, or visits $Locs$ at some intermediate step. Based on this characterization, we show that $\mathsf{ELTL}_{\mathsf{FT}}$ formulas are preserved if we move carefully chosen threads to the beginning of a steady schedule (intuitively, this corresponds to τ_1, and τ_2 from Theorem 6.3). Then, we replace the threads, one by one, by their representative schedules from Proposition 6.1, and append another representative schedule for the remainder of the schedule. In this way, we then obtain the representative schedules in Theorem 6.3(d).

Example 6.5. We consider the TA in Figure 2, and show how a schedule $\tau = (r_1, 1), (r_6, 1), (r_4, 1), (r_2, 1), (r_4, 1)$ applicable to σ_1, with $\tau(\sigma_1) = \sigma_2$ can be shortened. Figure 9 follows this example where τ is the upper schedule. Assume that $\mathsf{Cfgs}(\sigma_1, \tau) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, and that we want to construct a shorter schedule that produces a path that satisfies the same formula.

In our theory, subschedule $(r_1, 1), (r_4, 1)$ is a thread of σ_1 and τ for two reasons: (1) the counter of the starting local state of $(r_1, 1)$ is greater than 0, i.e., $\sigma_1.\boldsymbol{\kappa}[\ell_0] = 1$, and (2) it is a sequence of rules in the control flow of the threshold automaton, i.e., it starts from ℓ_0, then uses $(r_1, 1)$ to go to local state ℓ_2 and then $(r_4, 1)$ to arrive at ℓ_3. The intuition of (2) is that a thread corresponds to a process that executes the threshold automaton. Similarly, $(r_6, 1), (r_2, 1)$ and $(r_4, 1)$ are also threads of σ_1 and τ. In fact, we can show that each schedule can be decomposed into threads. Based on this, we analyze which local states are visited when a thread is executed.

Our formula $\mathsf{Cfgs}(\sigma_1, \tau) \models \boldsymbol{\kappa}[\ell_2] \neq 0$ talks about ℓ_2. Thus, we are interested in a thread that ends at ℓ_2, because after executing this thread, intuitively there will always be at least one process in ℓ_2, i.e., the counter $\boldsymbol{\kappa}[\ell_2]$ will be nonzero, as required. Such a thread will be moved to the beginning. We find that thread $(r_6, 1), (r_2, 1)$ meets this requirement. Similarly, we are also interested in a thread that starts from ℓ_2. Before we execute such a thread, at least one process must always be in ℓ_2, i.e., $\boldsymbol{\kappa}[\ell_2]$ will be nonzero. For this, we single out the thread $(r_4, 1)$, as it starts from ℓ_2.

Independently of the actual positions of these threads within a schedule, our condition $\boldsymbol{\kappa}[\ell_2] \neq 0$ is true *before* $(r_4, 1)$ starts, and *after* $(r_6, 1), (r_2, 1)$ ends. Hence, we move the thread $(r_6, 1), (r_2, 1)$ to the beginning, and obtain a schedule that ensures our condition in all visited configurations; cf. the lower schedule in Figure 9. Then we replace the thread $(r_6, 1), (r_2, 1)$, by a representative schedule from Proposition 6.1, and the remaining part $(r_1, 1), (r_4, 1), (r_4, 1)$, by another one. Indeed in our example, we could merge $(r_4, 1), (r_4, 1)$ into one accelerated transition $(r_4, 2)$ and obtain a schedule which is shorter than τ while maintaining $\boldsymbol{\kappa}[\ell_2] \neq 0$. ◁

7. Application of the Short Counterexample Property and Experimental Evaluation

7.1 SMT Encoding

We use the theoretical results from the previous section to give an efficient encoding of lasso-shaped executions in SMT with linear integer arithmetic. The definitions of counter systems in Section 2.2 directly tell us how to encode paths of the counter system. Definition 2.5 describes a configuration σ as tuple $(\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$, where each component is encoded as a vector of SMT integer variables. Then, given a path $\sigma_0, t_1, \sigma_1, \ldots, t_{k-1}, \sigma_{k-1}, t_k, \ldots \sigma_k$ of length k, by $\boldsymbol{\kappa}^i$, $\mathbf{g}^i$, and $\mathbf{p}^i$ we denote the values of the vectors that correspond to σ_i, for $0 \leq i \leq k$. As the parameter values do not change, we use one copy of the variables $\mathbf{p}$ in our SMT encoding. By $\boldsymbol{\kappa}^i_\ell$, for $1 \leq \ell \leq |\mathcal{L}|$, we denote the ℓth component of $\boldsymbol{\kappa}^i$, that is, the counter corresponding to the number of processes in local state ℓ after the ith iteration. Definition 2.5 also gives us the constraint on the initial states, namely:

$$\mathrm{init}(0) \equiv \sum_{\ell \in \mathcal{I}} \boldsymbol{\kappa}^0_\ell = N(\mathbf{p}) \wedge \sum_{\ell \notin \mathcal{I}} \boldsymbol{\kappa}^0_\ell = 0 \wedge \mathbf{g}^0 = \mathbf{0} \wedge RC(\mathbf{p}) \quad (4)$$

Example 7.1. The TA from Figure 2 has four local states ℓ_0, ℓ_1, ℓ_2, ℓ_3 among which ℓ_0 and ℓ_1 are the initial states. In this example, $N(\mathbf{p})$ is $n - f$, and the resilience condition requires that there are less than a third of the processes faulty, i.e., $n > 3t$. We obtain $\mathrm{init}(0) \equiv \boldsymbol{\kappa}^0_0 + \boldsymbol{\kappa}^0_1 = n - f \wedge \boldsymbol{\kappa}^0_2 + \boldsymbol{\kappa}^0_3 = 0 \wedge x^0 = 0 \wedge n > 3t \wedge t \geq f \wedge f \geq 0$. The constraint is in linear integer arithmetic. ◁

Further, Definition 2.8 encodes the transition relation. A transition is identified by a rule and an acceleration factor. A rule is identified by threshold guards $\varphi^{\leq}$ and $\varphi^{>}$, local states *from* and *to* between which processes are moved, and by $\mathbf{u}$, which defines the increase of shared variables. As according to Section 5 only a fixed number of transitions change the context and thus may change the evaluation of $\varphi^{\leq}$ and $\varphi^{>}$, we do not encode $\varphi^{\leq}$ and $\varphi^{>}$ for each rule. In fact, we check the guards $\varphi^{\leq}$ and $\varphi^{>}$ against a fixed number of configurations, which correspond to the cut points defined by the threshold guards. The acceleration factor δ is indeed the only variable in a transition, and the SMT solver has to find assignments

of these factors. Then this transition from the ith to the $(i+1)$th configuration is encoded using rule $r = (\mathit{from}, \mathit{to}, \varphi^{\le}, \varphi^{>}, \mathbf{u})$ as follows:

$$
\begin{aligned}
T(i,r) &\equiv \mathit{Move}(\mathit{from}, \mathit{to}, i) \wedge \mathit{IncShd}(\mathbf{u}, i) \qquad (5)\\
\mathit{Move}(\ell,\ell',i) &\equiv \ell \neq \ell' \rightarrow \boldsymbol{\kappa}^i_\ell - \boldsymbol{\kappa}^{i+1}_\ell = \delta^{i+1} = \boldsymbol{\kappa}^{i+1}_{\ell'} - \boldsymbol{\kappa}^i_{\ell'}\\
&\wedge \quad \ell = \ell' \rightarrow \left(\boldsymbol{\kappa}^i_\ell = \boldsymbol{\kappa}^{i+1}_\ell \wedge \boldsymbol{\kappa}^{i+1}_{\ell'} = \boldsymbol{\kappa}^i_{\ell'}\right)\\
&\wedge \quad \bigwedge_{s \in \mathcal{L}\setminus\{\ell,\ell'\}} \boldsymbol{\kappa}^i_s = \boldsymbol{\kappa}^{i+1}_s\\
\mathit{IncShd}(\mathbf{u}, i) &\equiv \mathbf{g}^{i+1} - \mathbf{g}^i = \delta^{i+1} \cdot \mathbf{u}
\end{aligned}
$$

Given a schedule τ, we encode in linear integer arithmetic the paths that follow this schedule from an initial state as follows:

$$E(\tau) \equiv \mathrm{init}(0) \wedge T(0, r_1) \wedge T(1, r_2) \wedge \dots$$

We can now ask the SMT solver for assignments of the parameters as well as the factors $\delta^1, \delta^2, \dots$ in order to check whether a path with this sequence of rules exists. Note that some factors can be equal to 0, which means that the corresponding rule does not have any effect (because no process executes it). If τ encodes a lasso shape, and the SMT solver reports a satisfying assignment, this assignment is a counterexample. If the SMT solver reports unsat on all lassos discussed in Section 5, then there does not exists a counterexample and the algorithm is verified.

Example 7.2. In Example 3.3 we have seen the fairness requirement ψ_{fair}, which is a property of a configuration that can be encoded as $\text{fair}(i) \equiv \boldsymbol{\kappa}^i_1 = 0 \wedge (x^i \geq t+1 \rightarrow \boldsymbol{\kappa}^i_0 = 0 \wedge \boldsymbol{\kappa}^i_1 = 0) \wedge (x^i \geq n - t \rightarrow \boldsymbol{\kappa}^i_0 = 0 \wedge \boldsymbol{\kappa}^i_2 = 0)$, which is a formula in linear integer arithmetic. Then, e.g., fair(5) encodes that the fifth configuration satisfies the predicate. Such state formulas can be added as conjunct to the formula $E(\tau)$ that encodes a path. ◁

As discussed in Sections 4 and 5 we have to encode lassos of the form $\vartheta \cdot \rho^\omega$ starting from an initial configuration σ. We immediately obtain a finite representation by encoding the fixed length execution $E(\vartheta \cdot \rho)$ as above, and adding the constraint that applying ρ returns to the start of the lasso loop, that is, $\vartheta(\sigma) = \rho(\vartheta(\sigma))$. In SMT this is directly encoded as equality of integer variables.

7.2 Generating the SMT Queries

The high-level structure of the verification algorithm is given in Figure 3 on page 6. In this section, we give the details of the procedure `check_one_order`, whose pseudo code is given in Figure 10. It receives as the input the following parameters: a threshold automaton TA, an $\mathsf{ELTL_{FT}}$ formula φ, a cut graph $\mathcal{G}$ of φ, a threshold graph $\mathcal{H}$ of TA, and a topological order $\prec$ on the vertices of the graph $\mathcal{G} \cup \mathcal{H}$.

The procedure `check_one_order` constructs SMT assertions about the configurations of the lassos that correspond to the order $\prec$. As explained in Section 7.1, an ith configuration is defined by the vectors of SMT variables $(\boldsymbol{\kappa}^i, \mathbf{g}^i, \mathbf{p})$. We use two global variables: the number fn of the configuration under construction, and the number fs of the configuration that corresponds to the loop start. Thus, with the expressions $\boldsymbol{\kappa}^{\mathsf{fn}}$ and $\mathbf{g}^{\mathsf{fn}}$ we refer to the SMT variables of the configuration whose number is stored in fn.

In the pseudocode in Figure 10, we call `SMT_assert(`$\boldsymbol{\kappa}^{\mathsf{fn}}$`,` $\mathbf{g}^{\mathsf{fn}}$`,` $\mathbf{p} \models \psi$`)` to add an assertion ψ about the configuration $(\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p})$ to the SMT query. Finally, the call `SMT_sat()` returns true, only if there is a satisfying assignment for the assertions collected so far. Such an assignment can be accessed with `SMT_model()` and gives the values for the configurations and acceleration factors, which together constitute a witness lasso.

The procedure `check_one_order` creates the assertions about the initial configurations. The assertions consist of: the assumptions init(0) about the initial configurations of the threshold automaton, the top-level propositional formula ψ_0, and the invariant propositional formula ψ_{k+1} that should hold from the initial configuration on. By writing `assume(`$\psi = \psi_0 \wedge \mathbf{F} \wedge \psi_1 \dots \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$`)`, we extract the subformulas of a canonical formula ψ (see Section 4.2). The procedure finds the minimal node in the order $\prec$ on the nodes of the graph $\mathcal{G} \cup \mathcal{H}$ and calls the procedure `check_node` for the initial node, the initial invariant ψ_{k+1}, and the empty context $\emptyset$.

The procedure `check_node` is called with a node v of the graph $\mathcal{G} \cup \mathcal{H}$ as a parameter. It adds assertions that encode a finite path and constraints on the configurations of this path. The finite path leads from the configuration that corresponds to the node v to the configuration that corresponds to v's successor in the order $\prec$.

1 **variables** fn, fs; *// the current configuration number and the loop start*
2 *// Try to find a witness lasso for: a threshold automaton* TA,
3 *// an* $\mathsf{ELTL_{FT}}$ *formula* φ*, a cut graph* $\mathcal{G}$*, a threshold graph* $\mathcal{H}$*, and*
4 *// a topological order* $\prec$ *on the nodes of* $\mathcal{G} \cup \mathcal{H}$*.*
5 **procedure** check_one_order(TA, φ, $\mathcal{G}$, $\mathcal{H}$, $\prec$):
6 fn := 0; fs := 0;
7 SMT_start(); *// start (or reset) the SMT solver*
8 assume($can(\varphi) = \psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \dots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$);
9 SMT_assert($\boldsymbol{\kappa}^0, \mathbf{g}^0, \mathbf{p} \models$ init$(0) \wedge \psi_0 \wedge \psi_{k+1}$); *// see Equation* 4
10 $v_0 := \min_\prec(\mathcal{V}_\mathcal{G} \cup \mathcal{V}_\mathcal{H})$; *// the minimal node w.r.t. the linear order* $\prec$
11 check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, v_0, ψ_{k+1}, $\emptyset$);
12
13 *// Try to find a witness lasso starting with the node* v *and the context* Ω*,*
14 *// while preserving the invariant* ψ_{inv}*.*
15 **recursive procedure** check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, v, ψ_{inv}, Ω):
16 **if not** SMT_sat() **then**:
17 **return** no_witness;
18 **case** (a) $v \in \mathcal{V}_\mathcal{G} \setminus \{\mathsf{loop_{start}}, \mathsf{loop_{end}}\}$:
19 find ψ s.t. $\langle \mathbf{F}\,\psi, v\rangle \in \mathcal{T}(\varphi)$; *// v labels a formula in the syntax tree*
20 assume($\psi = \psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \dots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$);
21 SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p} \models \psi_0$);
22 push_segment($\psi_{inv} \wedge \psi_{k+1}$);
23 $v' := \min_\prec(\mathcal{V}_\mathcal{G} \cup \mathcal{V}_\mathcal{H}) \cap \{w : v \prec w\}$; *// the next node after* v
24 check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, v', $\psi_{inv} \wedge \psi_{k+1}$, Ω);
25 **case** (b) $v \in \mathcal{V}_\mathcal{H} \setminus \{\mathsf{loop_{start}}, \mathsf{loop_{end}}\}$: *// v is a threshold guard*
26 **if** $v \in \Phi^{\mathrm{rise}}$ **then**: *// v is an unlocking guard, e.g.,* $x \geq t + 1 - f$
27 push_segment(ψ_{inv}); *// one rule unlocks v*
28 SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p} \models v$); *// v is unlocked*
29 push_segment(ψ_{inv}); *// execute all unlocked rules*
30 $v' := \min_\prec(\mathcal{V}_\mathcal{G} \cup \mathcal{V}_\mathcal{H}) \cap \{w : v \prec w\}$; *// the next node after* v
31 check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, v', ψ_{inv}, $\Omega \cup \{v\}$);
32 **else**: /* $v \in \Phi^{\mathrm{fall}}$*, e.g.,* $x < f$*, similar to the locking case: use* $\neg v$ */
33 **case** (c) $v = \mathsf{loop_{start}}$:
34 fs := fn; *// the loop starts at the current configuration*
35 push_segment(ψ_{inv}); *// execute all unlocked rules*
36 $v' := \min_\prec(\mathcal{V}_\mathcal{G} \cup \mathcal{V}_\mathcal{H}) \cap \{w : v \prec w\}$; *// the next node after* v
37 check_node($\mathcal{G}$, $\mathcal{H}$, $\prec$, v', ψ_{inv}, Ω);
38 **case** (d) $v = \mathsf{loop_{end}}$:
39 SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}} = \boldsymbol{\kappa}^{\mathsf{fs}} \wedge \mathbf{g}^{\mathsf{fn}} = \mathbf{g}^{\mathsf{fs}}$); *// close the loop*
40 **if** SMT_sat() **then**:
41 **return** witness(SMT_model())
42
43 *// Encode a segment of rules as prescribed by* [42] *and Theorems* 6.3–6.4*.*
44 **procedure** push_segment(ψ_{inv}):
45 *// find the number of schedules to repeat in (d) of Theorems* 6.3, 6.4
46 nrepetitions := compute_repetitions(ψ_{inv});
47 $r_1, \dots, r_k$:= compute_rules(Ω); *// use* $\mathsf{sschema}_\Omega$ *from* [42]
48 **for** _ **from** 1 **to** nrepetitions:
49 **for** j **from** 1 **to** k:
50 SMT_assert($\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p} \models \psi_{inv}$);
51 SMT_assert($T(\mathsf{fn}, r_j)$); *// modify the counters as in Equation* 5
52 fn := fn + 1; *// move to the next configuration*

Figure 10. Checking one topological order with SMT.

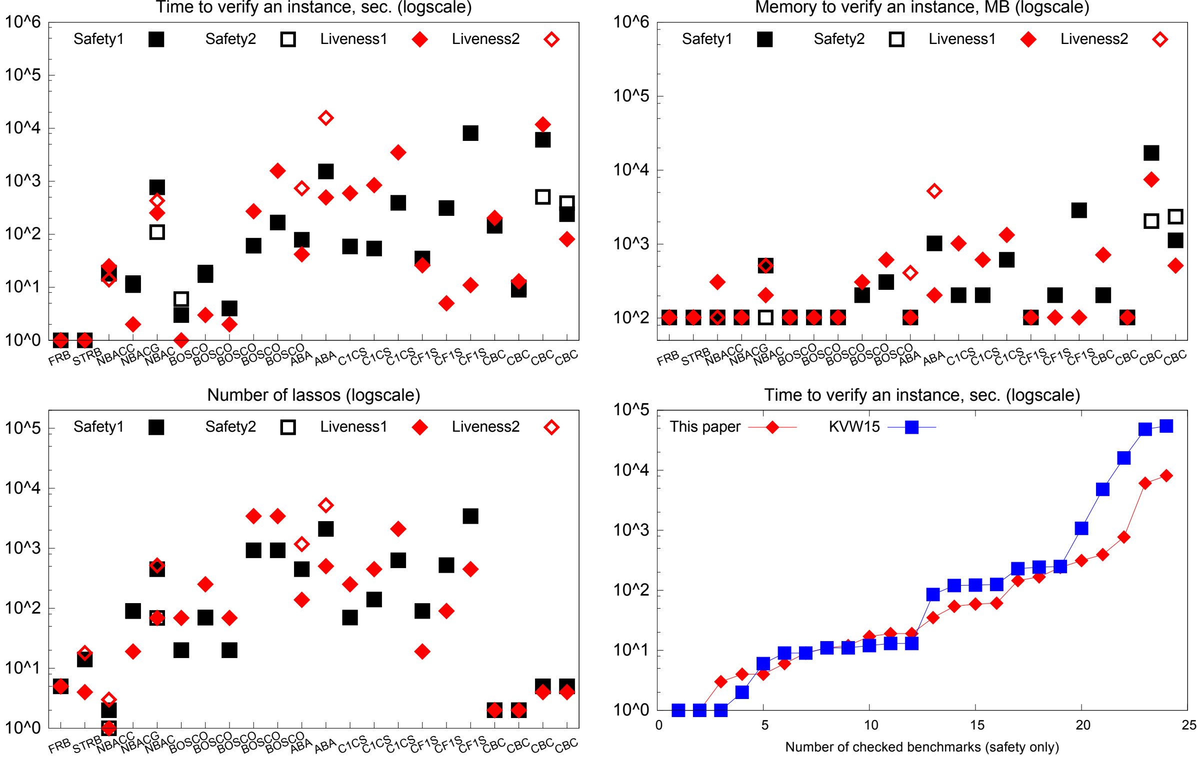


Figure 11. The plots summarize the following results of running our implementation on all benchmarks: used time in seconds (top left), used memory in megabytes (top right), the number of checked lassos (bottom left), time used both by our implementation and [42] to check *safety only* (bottom right). Several occurrences of the same benchmark correspond to different cases, such as $f > 1$, $f = 1$, and $f = 0$. Symbols ■ and □ correspond to the safety properties of each benchmark, while symbols ♦ and ◊ correspond to the liveness properties.

The constraints depend on v's origin: (a) v labels a formula $\mathsf{F}\,\psi$ in the syntax tree of φ, (b) v carries a threshold guard from the set $\Phi^{\text{rise}} \cup \Phi^{\text{fall}}$, (c) v denotes the loop start, or (d) v denotes the loop end. In case (a), we add an SMT assertion that the current configuration satisfies the propositional formula $prop(\psi)$ (line 21), and add a sequence of rules that leads to v's successor while maintaining the invariants ψ_{inv} of the preceding nodes and the v's invariant ψ_{k+1} (line 22). In case (b), in line 27, we add a sequence of rules, one of which should unlock (resp. lock) the threshold guard in $v \in \Phi^{\text{rise}}$ (resp. $v \in \Phi^{\text{fall}}$). Then, in line 29, we add a sequence of rules that leads to a configuration of v's successor. All added configurations are required to satisfy the current invariant ψ_{inv}. As the threshold guard in v is now unlocked (resp. locked), we include the guard (resp. its negation) in the current context Ω. In case (c), we store the current configuration as the loop start in the variable fs and, as in (a) and (b), add a sequence of rules leading to v's successor. Finally, in case (d), we should have reached the ending configuration that coincides with the loop start. To this end, in line 39, we add the constraint that forces the counters of both configurations to be equal. At this point, all the necessary SMT constraints have been added, and we call `SMT_sat` to check whether there is an assignment that satisfies the constraints. If there is one, we report it as a lasso witnessing the $\mathsf{ELTL_{FT}}$-formula φ that consists of: the concrete parameter values, the values of the counters and shared variables for each configuration, and the acceleration factors. Otherwise, we report that there is no witness lasso for the formula φ.

The procedure `push_segment` constructs a sequence of currently unlocked rules, as in the case of reachability [42]. However, this sequence should be repeated several times, as required by Theorems 6.3 and 6.4. Moreover, the freshly added configurations are required to satisfy the current invariant ψ_{inv}.

7.3 Experiments

We extended the tool ByMC [42] with our technique and conducted experiments[2] with the freely available benchmarks from [42]: folklore reliable broadcast (FRB) [14], consistent broadcast (STRB) [64], asynchronous Byzantine agreement (ABA) [11], condition-based consensus (CBC) [52], non-blocking atomic commitment (NBAC and NBACC [61] and NBACG [37]), one-step consensus with zero degradation (CF1S [21]), consensus in one communication step (C1CS [12]), and one-step Byzantine asynchronous consensus (BOSCO [63]). These threshold-guarded fault-tolerant distributed algorithms are encoded in a parametric extension of Promela.

Negations of the safety and liveness specifications of our benchmarks — written in $\mathsf{ELTL_{FT}}$ — follow three patterns: unsafety $\mathsf{E}\,(p \wedge \mathsf{F}\,q)$, non-termination $\mathsf{E}\,(p \wedge \mathsf{G}\,\mathsf{F}\,r \wedge \mathsf{G}\,q)$, and non-response $\mathsf{E}\,(\mathsf{G}\,\mathsf{F}\,r \wedge \mathsf{F}\,(p \wedge \mathsf{G}\,q))$. The propositions p, q, and r follow the syntax of *pform* (cf. Table 1), e.g., $p \equiv \bigwedge_{\ell \in Locs_1} \kappa[\ell] = 0$ and $q \equiv \bigvee_{\ell \in Locs_2} \kappa[\ell] \neq 0$ for some sets of locations $Locs_1$ and $Locs_2$.

The results of our experiments are summarized in Figure 11. Given the properties of the distributed algorithms found in the literature, we checked for each benchmark one or two safety properties (depicted with ■ and □) and one or two liveness properties (depicted with ♦ and ◊). For each benchmark, we display the running times and the memory used together by ByMC and the SMT

[2] The details on the experiments and the artifact are available at: `http://forsyte.at/software/bymc/popl17-artifact`

solver Z3 [20], as well as the number of exercised lasso shapes as discussed in Section 5.

For safety properties, we compared our implementation against the implementation of [42]. The results are summarized the bottom right plot in Figure 11, which shows that there is no clear winner. For instance, our implementation is 170 times faster on BOSCO for the case $n > 5t$. However, for the benchmark ABA we experienced a tenfold slowdown. In our experiments, attempts to improve the SMT encoding for liveness usually impaired safety results.

Our implementation has verified safety and liveness of all ten parameterized algorithms in less than a day. Moreover, the tool reports counterexamples to liveness of CF1S and BOSCO exactly for the cases predicted by the distributed algorithms literature, i.e., when there are not enough correct processes to reach consensus in one communication step. Noteworthy, liveness of only the two simplest benchmarks (STRB and FRB) had been automatically verified before [40].

8. Conclusions

Parameterized verification approaches the problem of verifying systems of thousands of processes by proving correctness for all system sizes. Although the literature predominantly deals with safety, parameterized verification for liveness is of growing interest, and has been addressed mostly in the context of programs that solve mutual exclusion or dining philosophers [4, 30, 31, 59]. These techniques do not apply to fault-tolerant distributed algorithms that have arithmetic conditions on the fraction of faults, threshold guards, and typical specifications that evaluate a global system state.

Parameterized verification is in general undecidable [3]. As recently surveyed by Bloem et al. [9], one can escape undecidability by restricting, e.g., communication semantics, local state space, the local control flow, or the temporal logic used for specifications. Hence, we make explicit the required restrictions. On the one hand, these restrictions still allow us to model fault-tolerant distributed algorithms and their specifications, and on the other hand, they give rise to a practical verification method. The restrictions are on the local control flow (loops) of processes (Section 2.1), as well as on the temporal operators and propositional formulas (Section 3). We conjecture that lifting these restrictions quite quickly leads to undecidability again. In addition, we justify our restrictions with the considerable number of benchmarks [11, 12, 14, 21, 37, 52, 61, 63, 64] that fit into our fragment, and with the convincing experimental results from Figure 11.

Our main technical contribution is to combine and extend several important techniques: First, we extend the ideas by Etessami et al. [29] to reason about shapes of infinite executions of lasso shape. These executions are counterexample candidates. Then we extend reductions introduced by Lipton [50] to deal with $\mathsf{ELTL_{FT}}$ formulas. (Techniques that extend Lipton's in other directions can be found in [19, 22, 25, 34, 44, 47].) Our reduction is specific to threshold guards which are typical for fault-tolerant distributed algorithms and are found in domain-specific languages. Using on our reduction we apply acceleration [6, 44] in order to arrive at our short counterexample property.

Our short counterexample property implies a completeness threshold, that is, a bound b that ensures that if no lasso of length up to b is satisfying an $\mathsf{ELTL_{FT}}$ formula, then there is no infinite path satisfying this formula. For linear temporal logic with the F and G operators, Kroening et al. [46] prove bounds on the completeness thresholds on the level of Büchi automata. Their bound involves the recurrence diameter of the transition systems, which is prohibitively large for counter systems. Similarly, the general method to transfer liveness with fairness to safety checking by Biere et al. [8] leads to an exponential growth of the diameter, and thus to too large values of b. Hence, we decided to conduct an analysis on the level of threshold automata, accelerated counter systems, and a fragment of the temporal logic, which allows us to exploit specifics of the domain, and get bounds that can be used in practice.

Acceleration has been applied for parameterized verification by means of regular model checking [1, 10, 58, 62]. As noted by Fisman et al. [33], to verify fault-tolerant distributed algorithms, one would have to intersect the regular languages that describe sets of states with context-free languages that enforce the resilience condition (e.g., $n > 3t$). Our approach of reducing to SMT handles resilience conditions naturally in linear integer arithmetic.

There are two reasons for our restrictions in the temporal logic: On one hand, in our benchmarks, there is no need to find counterexamples that contain a configuration that satisfies $\boldsymbol{\kappa}[\ell] = 0 \vee \boldsymbol{\kappa}[\ell'] = 0$ for some $\ell, \ell' \in \mathcal{L}$. One would only need such a formula to specify requirement that at least one process is at location ℓ and at least one process is at location ℓ' (the disjunction would be negated in the specification), which is unnatural for fault-tolerant distributed algorithms. On the other hand, enriching our logic with $\bigvee_{i \in Locs} \boldsymbol{\kappa}[i] = 0$ allows one to express tests for zero in the counter system, which leads to undecidability [9]. For the same reason, we avoid disjunction, as it would allow one to indirectly express test for zero: $\boldsymbol{\kappa}[\ell] = 0 \vee \boldsymbol{\kappa}[\ell'] = 0$.

The restrictions we put on threshold automata are justified from a practical viewpoint of our application domain, namely, threshold-guarded fault-tolerant algorithms. We assumed that all the cycles in threshold automata are simple (while the benchmarks have only self-loops or cycles of length 2). As our analysis already is quite involved, these restrictions allow us to concentrate on our central results without obfuscating the notation and theoretical results. Still, from a theoretical viewpoint it might be interesting to relax the restrictions on cycles in the future.

More generally, these restrictions allowed us to develop a completely automated verification technique. In general, there is a trade-off between degree of automation and generality. Our method is completely automatic, but our input language cannot compete in generality with mechanized proof methods that rely heavily on human expertise, e.g., IVY [55], Verdi [68], IronFleet [38], TLAPS [16].

References

[1] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *CAV*, LNCS, pages 305–318, 1998.

[2] F. Alberti, S. Ghilardi, and E. Pagani. Counting constraints in flat array fragments. In *IJCAR*, volume 9706 of *LNCS*, pages 65–81, 2016.

[3] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.

[4] M. F. Atig, A. Bouajjani, M. Emmi, and A. Lal. Detecting fair non-termination in multithreaded programs. In *CAV*, pages 210–226, 2012.

[5] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[6] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.

[7] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: a framework for implementing fault-tolerant distributed algorithms. In *DSN*, pages 1–8, 2013.

[8] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2): 160–177, 2002.

[9] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[10] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, LNCS, pages 372–386, 2004.

[11] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[12] F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *PaCT*, volume 2127 of *LNCS*, pages 42–50, 2001.

[13] E. R. Canfield and S. G. Williamson. A loop-free algorithm for generating the linear extensions of a poset. *Order*, 12(1):57–75, 1995.

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[15] B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *IJSI*, 3(2–3):273–303, 2009.

[16] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, volume 6173 of *LNCS*, pages 142–148, 2010.

[17] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[18] E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In *TACAS'08/ETAPS'08*, pages 33–47. Springer, 2008.

[19] E. Cohen and L. Lamport. Reduction in TLA. In *CONCUR*, volume 1466 of *LNCS*, pages 317–331, 1998.

[20] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. 2008.

[21] D. Dobre and N. Suri. One-step consensus with zero-degradation. In *DSN*, pages 137–146, 2006.

[22] T. W. Doeppner. Parallel program correctness through refinement. In *POPL*, pages 155–169, 1977.

[23] C. Drăgoi, T. A. Henzinger, and D. Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415, 2016.

[24] C. Drăgoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.

[25] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[26] E. Emerson and K. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.

[27] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE, 2003.

[28] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.

[29] K. Etessami, M. Y. Vardi, and T. Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.

[30] Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with invisible ranking. *STTT*, 8(3):261–279, 2006.

[31] A. Farzan, Z. Kincaid, and A. Podelski. Proving liveness of parameterized programs. In *LICS*, pages 185–196, 2016.

[32] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[33] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, volume 4963 of *LNCS*, pages 315–331. Springer, 2008.

[34] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*, 31(4):275–291, 2005.

[35] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[36] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *Formal Methods for Executable Software Models*, LNCS, pages 122–171. Springer, 2014.

[37] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15 (1):17–25, 2002.

[38] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

[39] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[40] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.

[41] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN PLDI*, pages 179–188, 2007.

[42] I. Konnov, H. Veith, and J. Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

[43] I. Konnov, M. Lazić, H. Veith, and J. Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. *CoRR*, abs/1608.05327, 2016. URL `http://arxiv.org/abs/1608.05327`.

[44] I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 2016. Accepted manuscript available online: 10-MAR-2016. `http://dx.doi.org/10.1016/j.ic.2016.03.006`.

[45] I. Konnov, H. Veith, and J. Widder. What you always wanted to know about model checking of fault-tolerant distributed algorithms. In *PSI 2015, Revised Selected Papers*, volume 9609 of *LNCS*, pages 6–21. Springer, 2016.

[46] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, and J. Worrell. Linear completeness thresholds for bounded model checking. In *CAV*, volume 6806 of *LNCS*, pages 557–572, 2011.

[47] L. Lamport and F. B. Schneider. Pretending atomicity. Technical Report 44, SRC, 1989.

[48] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370, 2016.

[49] P. Lincoln and J. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *FTCS*, pages 402–411, 1993.

[50] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[51] B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. I. *Acta Informatica*, 21(2): 125–169, 1984.

[52] A. Mostéfaoui, E. Mourgaya, P. R. Parvédy, and M. Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

[53] Netflix. 5 lessons we have learned using AWS. 2010. retrieved on Nov. 7, 2016. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`.

[54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–320, 2014.

[55] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630, 2016.

[56] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[57] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making fast consensus generally faster. In *DSN*, pages 156–167, 2016.

[58] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV*, LNCS, pages 328–343, 2000.

[59] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0,1,\infty)$- counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 93–111. 2002.

[60] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *ECEASST*, 72, 2015.

[61] M. Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *HASE*, pages 209–214, 1997.

[62] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes in Theoretical Computer Science*, 149(1):79–96, 2006.

[63] Y. J. Song and R. van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

[64] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

[65] TLA. TLA+ toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html`.

[66] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331, 1986.

[67] K. von Gleissenthall, N. Bjørner, and A. Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, pages 599–613, 2016.

[68] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

Part III

Parameterized Synthesis of Threshold-Guarded Distributed Algorithms

Chapter 7

Synthesis of Distributed Algorithms with Parameterized Threshold Guards

Marijana Lazić, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of Distributed Algorithms with Parameterized Threshold Guards. OPODIS, pp. 32:1-32:20, 2017.

DOI: http://dx.doi.org/10.4230/LIPIcs.OPODIS.2017.32

Synthesis of Distributed Algorithms with Parameterized Threshold Guards*

Marijana Lazić[1], Igor Konnov[2], Josef Widder[3], and Roderick Bloem[4]

1 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`lazic@forsyte.at`
2 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`konnov@forsyte.at`
3 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`widder@forsyte.at`
4 **TU Graz, Inffeldgasse 16a/II, 8010 Graz, Austria**
`roderick.bloem@iaik.tugraz.at`

Abstract

Fault-tolerant distributed algorithms are notoriously hard to get right. In this paper we introduce an automated method that helps in that process: the designer provides specifications (the problem to be solved) and a sketch of a distributed algorithm that keeps arithmetic details unspecified. Our tool then automatically fills the missing parts.

Fault-tolerant distributed algorithms are typically parameterized, that is, they are designed to work for any number n of processes and any number t of faults, provided some resilience condition holds; e.g., $n > 3t$. In this paper we automatically synthesize distributed algorithms that work for *all* parameter values that satisfy the resilience condition. We focus on threshold-guarded distributed algorithms, where actions are taken only if a sufficiently large number of messages is received, e.g., more than t or $n/2$. Both expressions can be derived by choosing the right values for the coefficients a, b, and c, in the sketch of a threshold $a \cdot n + b \cdot t + c$. Our method takes as input a sketch of an asynchronous threshold-based fault-tolerant distributed algorithm — where the guards are missing exact coefficients — and then iteratively picks the values for the coefficients.

Our approach combines recent progress in parameterized model checking of distributed algorithms with counterexample-guided synthesis. Besides theoretical results on termination of the synthesis procedure, we experimentally evaluate our method and show that it can synthesize several distributed algorithms from the literature, e.g., Byzantine reliable broadcast and Byzantine one-step consensus. In addition, for several new variations of safety and liveness specifications, our tool generates new distributed algorithms.

1998 ACM Subject Classification F.3.1 [*Logic and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.4.5 [*Software*]: Operating systems: Fault-tolerance, Verification

Keywords and phrases fault-tolerant distributed algorithms – Byzantine faults – parameterized model checking – program synthesis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.32

* Supported by: the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403, S11405, and S11406), project PRAVDA (P27722), and Doctoral College LogiCS (W1255-N23); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103). The computational results presented have been achieved [in part] using the Vienna Scientific Cluster (VSC).

© Marijana Lazić, Igor Konnov, Josef Widder, Roderick Bloem;
licensed under Creative Commons License CC-BY
21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão; Article No. 32; pp. 32:1–32:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Design and implementation of parameterized fault-tolerant distributed systems are error-prone tasks. There is a mature *theory* regarding mathematical proof methods, which found their way into formal frameworks like I/O Automata [24] and TLA+ [22]. Recent approaches [17, 23, 31] provide tool support to establish correctness of implementations, by manually constructing proofs with an interactive theorem prover. Although, if successful, this approach provides a machine-checkable proof [9, 4], it requires huge manual efforts from the user. A logic for distributed consensus algorithms in the HO Model [8] was introduced in [13], which allows one to automatically check the invariants (for safety) and ranking functions (for liveness), that is, the manual effort is reduced to finding right invariants and ranking functions. Model checking of distributed algorithms promises a higher degree of automation. For consensus algorithms in the HO Model, the results of [25] reduce the verification to checking small systems of five or seven processes. For the asynchronous model, an efficient model checking technique for threshold-guarded distributed algorithms was introduced in [20, 19]. Notably, this technique verifies both safety and liveness properties. In all these methods, the user has to produce an implementation (or design), and the goal is to check (using techniques that vary in the degree of automation) whether this implementation satisfies a given specification.

In this paper we explore synthesis as it promises even more automation. The user just provides required properties and a sketch of an asynchronous algorithm, and our tool automatically finds a correct distributed algorithm. In this way we generate new fault-tolerant algorithms that are *correct by construction*. In our experiments we first focus on existing specifications [29, 7, 30, 28] from the literature, in order to be able to compare the output of our tool with known algorithms. We then give new variations of safety and liveness specifications, and our tool generates new distributed algorithms for them.

Parametrized synthesis. Similar to the verification approaches above, we are interested in the parameterized version of the problem: Rather than synthesizing a distributed algorithm that consists of, say, four processes and tolerates one fault, our goal is to synthesize an algorithm that works for n processes, out of which t may fail, for all values of n and t that satisfy a resilience condition, e.g., $n > 3t$. This is in contrast to recent work on synthesis of fault-tolerant distributed algorithms [16, 15, 14] that requires the user to fix the number of processes; typically to some $n < 10$. In some special cases, manual arguments or cut-off theorems generalize synthesis results for small systems to parameterized ones [5, 12, 26]. However, similar to parameterized verification [2, 6], the parameterized synthesis problem is in general undecidable [18]. As in the parameterized verification approach of [19], we will therefore limit ourselves to a specific class of distributed algorithms, namely, *threshold-guarded distributed algorithms*. These thresholds are arithmetic expressions over parameters, e.g., $n/2$, and determine for how many messages processes should wait (a majority in the example).

More specifically, the user provides as input a distributed algorithm with holes as in Figure 1: The user defines the control flow, and keeps the threshold expressions — noted as τ_{0toSE} and τ_{AC} in the figure — unspecified. As pseudo code has no formal semantics, it cannot be used as a tool input. Rather, our tool takes as input a sketch threshold automaton.

▶ **Example 1.** Figure 1 is a pseudo code representation of the input, and Figure 2 shows the corresponding sketch threshold automaton; they are related as follows. The initial locations ℓ_0 and ℓ_1 of the sketch threshold automaton in Figure 2 correspond to initial states in Figure 1 where *myval* is equal to 0 and 1, respectively. Edges are labeled by $g \mapsto \texttt{act}$, where expression g is a threshold guard, and the action `act` may increment a shared variable.

```
Code of a correct process i:
var myval_i ∈ {0,1}
var accept_i ∈ {false, true} ← false

while true do (in one step)
 if myval_i = 1
   and not sent ECHO before
 then send ECHO to all

 if received ECHO from ≥ τ_0toSE
    distinct processes
   and not sent ECHO before
 then send ECHO to all

 if received ECHO from ≥ τ_AC
    distinct processes
 then accept_i ← true
od
```

Figure 1 A single-round version of the reliable broadcast algorithm [29] with holes

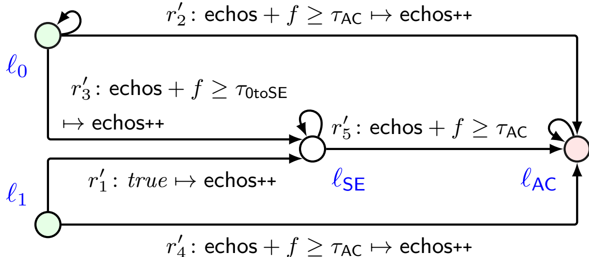


Figure 2 A sketch threshold automaton

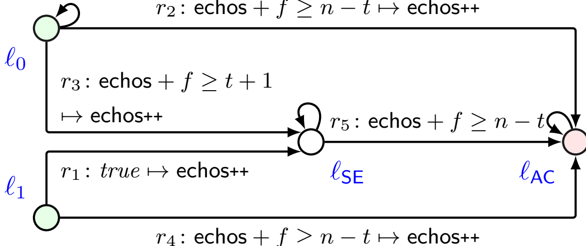


Figure 3 A synthesized threshold automaton

The action echos++ corresponds to the pseudo-code statement: `send <echo> to all`. (The message buffers are replaced by a shared variable that is increased whenever a message is sent. This typically can be done for algorithms that only count messages, and do not distinguish the senders. For instance, a bisimulation between models with message buffers and shared variables was proven in [21].) By going to local state ℓ_{SE}, a process records that it has sent *echo*. Finally, by going to the local state ℓ_{AC}, a process records that it has set accept to true. The "$+f$" terms in threshold guards model that messages from up to f Byzantine processes may be received ($f \leq t$), while we model only the $n - f$ correct processes explicitly.

We use self loops to capture the behavior that processes may take arbitrarily many steps before receiving a message sent to them, that is, asynchronous communication. For instance, the self loop in ℓ_0 allows processes to stay in ℓ_0 even if the guards of the other outgoing edges evaluate to true. That every message from a correct process is eventually received is a fairness constraint and is therefore not part of the threshold automaton, but is captured in the specifications. For instance, such a fairness constraint would be that if echos $\geq n - t$, then every process must eventually leave location ℓ_0. (In the fairness constraint, the "$+f$" does not appear, because messages from faulty processes are not guaranteed to arrive.)

The "holes" τ_{0toSE} and τ_{AC} in Figure 2 are the missing thresholds, which should be linear combination of the parameters n and t. Therefore τ_{0toSE} has the form $?_1 \cdot n + ?_2 \cdot t + ?_3$, and τ_{AC} has the form $?_4 \cdot n + ?_5 \cdot t + ?_6$. The unknown coefficients $?_i$, for $1 \leq i \leq 6$, have to be found by the synthesis tool. One solution is $?_1 = 0$, $?_2 = 1$, $?_3 = 1$, $?_4 = 1$, $?_5 = -1$, and $?_6 = 0$, that is, $\tau_{0toSE} = t + 1$ and $\tau_{AC} = n - t$. This solution is depicted in Figure 3. ◁

In addition to a sketch threshold automaton, the user has to provide a specification, that is, safety and liveness properties the distributed algorithm should satisfy. Based on these inputs, our tool generates the required coefficients, that is, a threshold automaton as in Figure 3. The synthesis approach of this paper is enabled by a recent advance [19] in parameterized model checking of *safety and liveness* properties of distributed algorithms.

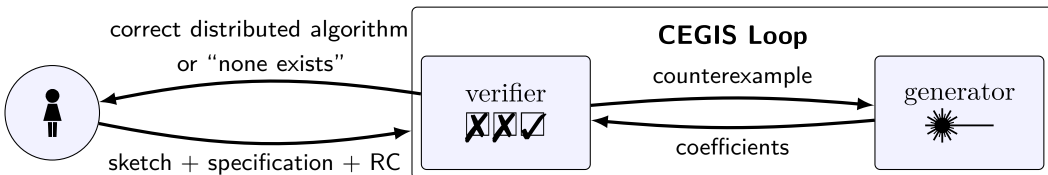


Figure 4 The synthesis loop implemented in this paper.

Existing model checking engine. The central idea of the verification approach in [19] is to formalize a distributed algorithm as counter system defined by a threshold automaton. A state of a counter system records how many processes are in which local state (e.g., $\ell_0, \ell_1, \ell_{\mathsf{SE}}, \ell_{\mathsf{AC}}$). A transition checks the guard and decreases or increases the related process counters. A transition can be written as a set of constraints in linear integer arithmetic LIA. (Threshold guards with rational coefficients, e.g., $\mathsf{echos} > \frac{n}{2}$, can be converted to integer constraints, e.g., $2 \cdot \mathsf{echos} > n$.) Thus, one can check for existence of specific executions by using SMT solvers, which extend SAT solvers (for Boolean satisfiability) with first-order theories, in our case, LIA. As shown in [20, 19], resilience conditions, executions of threshold-guarded distributed algorithms, and specifications can be encoded as logical formulas, whose satisfiability can be checked by solvers such as Z3 [11] and CVC4 [3]. In particular, the queries used in [20, 19] correspond to counterexamples to a specification: If the SMT solver finds all queries to be unsatisfiable, the distributed algorithm is correct. Otherwise, if a query is satisfiable, the SMT solver outputs a satisfying assignment, that is an error trace, called *counterexample*.

The synthesis approach of this paper. Figure 4 gives an overview of our method that takes as input (i) a sketch of a distributed algorithm, (ii) a set of safety and liveness specifications, and (iii) a resilience condition like $n > 3t$, and produces as output a correct distributed algorithm, or informs the user that none exist.

We follow the CEGIS approach to synthesis [1], which proceeds in a refinement loop. Roughly speaking, the verifier starts by picking default values for the missing coefficients — e.g., a vector of zeroes — and checks whether the algorithm is correct with these coefficients. Typically this is not the case and the *verifier* produces a counterexample. By automatically analyzing this counterexample, the *generator* learns constraints on the coefficients that are known to produce counterexamples. The generator gives these constraints to an SMT solver that generates new values for the coefficients, which are used in a new verifier run. If the verifier eventually reports that the current coefficients induce a correct distributed algorithm, we output this algorithm. The theory from [19] then implies correctness of the algorithm.

Termination of synthesis. The remaining theoretical problem that we address in this paper is termination of the refinement loop: In principle, the generator can produce infinitely many vectors of coefficients. In case there is no solution (which is typically the case in Byzantine fault tolerance if $n \leq 3t$), the naïve approach from the previous paragraph does not terminate, unless we restrict the guards to "reasonable" values. In this paper, we require the guards to lie in the interval $[0, n]$. We call such guards *sane*. For instance, although syntactically the expressions $\mathsf{echos} \leq -42n$ and $\mathsf{echos} > 2n$ are threshold guards, they are not sane, while $\mathsf{echos} \geq t + 1$ is sane. We mathematically prove that all sane guards of a specific structure have coefficients within a hyperrectangle. We call this hypperrectangle a *sanity box*, and prove that its boundaries depend only on the resilience condition. Within the sanity box,

there is only a finite number of coefficients, if we restrict them to integers or rationals with a fixed denominator. We thus obtain a finite search space and a completeness result for the synthesis loop.

Safety, liveness, and the fraction of faults. We consider the conjunction of *safety* and *liveness* specifications, as these specifications in isolation typically have trivial solutions; e.g., "do nothing" is always safe. If just given a safety specification, our tool generates thresholds like n for all guards, which leads to all guards evaluating to false initially. Hence, no action can ever be taken, which is a valid solution if liveness is not required.

Besides, our tool treats resilience conditions precisely. On the one hand, given the sketch from Figure 2, and the resilience condition $n > 3t$, in a few seconds our tool generates the threshold automaton in Figure 3. On the other hand, in the case of $n \geq 3t$, our tool reports (also within seconds) that no such algorithm exists, which in fact constitutes an automatically generated impossibility result for sane thresholds and a fixed sketch.

Experimental evaluation. We extended the tool ByMC [20] with our technique and conducted experiments based on the freely available benchmarks from [20]: folklore reliable broadcast [7], consistent broadcast [29, 30], and one-step Byzantine asynchronous consensus BOSCO [28]. For these benchmarks, we replaced the threshold guards by threshold guards with holes. By experimental evaluation, we show that our method can be used to generate coefficients even for quite intricate fault-tolerant distributed algorithms that tolerate Byzantine faults. In particular BOSCO proved to be a hard instance. It has to satisfy constraints derived from different safety an liveness specifications under different resilience conditions $n > 3t$, $n > 5t$, and $n > 7t$. Our tool is able to derive the three different threshold guards the algorithm requires. Finally, we give variations of specifications, and synthesize distributed algorithms from them that have not been produced before.

2 Modelling Threshold-Guarded Distributed Algorithms

Threshold-guarded algorithms are formalized by threshold automata. We recall the notions of threshold automata [19] and introduce the new concept of sketches. As usual, $\mathbb{N}_0$ is the set of natural numbers including 0, and $\mathbb{Q}$ is the set of rational numbers. The set Π is a finite set of *parameter variables* that range over $\mathbb{N}_0$. Typically, Π consists of three variables: n for the total number of processes, f for the number of actual faults in a run, and t for an upper bound on f. The parameter variables from Π are usually restricted to admissible combinations by a formula that is called a *resilience condition*, e.g., $n > 3t \wedge t \geq f \geq 0$. The set Γ is a finite set that contains *shared variables* that store the number of distinct messages sent by distinct (correct) processes, the variables in Γ also range over $\mathbb{N}_0$. In the example in Figure 3, $\Gamma = \{\mathsf{echos}\}$. For the variables from Γ, we will use names echos, x, y, etc.

For a set of variables V, a function $\nu : V \to \mathbb{Q}$ is called *an assignment*; its domain V is denoted with $dom(\nu)$. In this paper, we use Φ, Ψ, and Θ for first-order logic (FOL) formulas; e.g., when encoding linear integer constraints in SMT. For a FOL formula Φ, we write $free(\Phi)$ for the set of Φ's free variables, that is, the variables not bound with a quantifier. (For convenience, we assume that quantified variables have unique names and they are different from the names of the free variables.) Given an assignment $\nu : V \to \mathbb{Q}$ and a FOL formula Φ, we define a *substitution* $\Phi[\nu]$ as a FOL formula that is obtained from Φ by replacing all the variables from $V \cap free(\Phi)$ with their values in ν.

To introduce sketches of threshold automata — such as in Figure 2 — we define unknowns such as $\mathbf{?}_1$. The set U is a finite set of *unknowns* that range over $\mathbb{Q}$. For the variables from U, we use the names $\mathbf{?}_1$, $\mathbf{?}_2$, etc. We denote the rational values of unknowns with a, b, c, etc.

Generalized threshold guards, or just *guards*, are defined according to the grammar:

$\mathit{Guard} ::= \mathit{Shared} \geq \mathit{LinForm} \mid \mathit{Shared} < \mathit{LinForm}$	$\mathit{Shared} ::= \langle\text{variable from } \Gamma\rangle$
$\mathit{LinForm} ::= \mathit{FreeCoeff} \mid \mathit{Prod} \mid \mathit{Prod} + \mathit{LinForm}$	$\mathit{Param} ::= \langle\text{a variable from } \Pi\rangle$
$\mathit{FreeCoeff} ::= \mathit{Rat} \mid \mathit{Unknown}$	$\mathit{Unknown} ::= \langle\text{a variable from } U\rangle$
$\mathit{Prod} ::= \mathit{Rat} \times \mathit{Param} \mid \mathit{Unknown} \times \mathit{Param}$	$\mathit{Rat} ::= \langle\text{a rational from } \mathbb{Q}\rangle$

For convenience, we assume that every parameter appears in *LinForm* at most once. Let $\bar{\pi}$ denote the vector $(\pi_1, \ldots, \pi_{|\Pi|}, 1)$ that contains all the parameter variables from Π in a fixed order as well as number 1 as the last element. Then, every generalized guard can be written in one of the two following forms $x \geq \bar{u} \cdot \bar{\pi}^\intercal$ or $x < \bar{u} \cdot \bar{\pi}^\intercal$, where x is a shared variable from Γ, and $\bar{u}$ is a vector of elements from $U \cup \mathbb{Q}$. When a parameter does not appear in a generalized guard, its corresponding component in $\bar{u}$ equals zero. We say that a guard is a *sketch guard* if its vector $\bar{u}$ contains a variable from U. A guard that is not a sketch guard is called a *fixed guard.* Previous work [19] was only concerned with fixed guards.

Since threshold guards are a special case of FOL formulas, we can apply substitutions to them. For instance, given an assignment $\nu : U \to \mathbb{Q}$ and a threshold guard g, the substitution $g[\nu]$ replaces every occurence of an unknown $\mathbf{?}_i \in U$ in g with the rational $\nu(\mathbf{?}_i)$.

Threshold automata, denoted by TA, are edge-labeled graphs, where vertices are called *locations*, and edges are called *rules.* Rules are labeled by $g \mapsto \texttt{act}$, where expression g is a fixed threshold guard, and the action `act` may increment a shared variable. We define *generalized threshold automata* GTA, in the same way as threshold automata, with the only difference that expressions g in the edge labeling are generalized threshold guards. If all generalized guards in a GTA are fixed, then that GTA is a TA. If at least one of the edges of a GTA is labeled by a sketch guard, then we call this automaton a *sketch threshold automaton*, and we denote it by STA. Given an STA and an assignment $\nu : U \to \mathbb{Q}$, we obtain a threshold automaton STA$[\nu]$ by applying substitution $g[\nu]$ to every sketch guard g in STA.

Counter systems. Executions of threshold automata are formalized as counter systems. Since processes just wait for messages until a threshold is reached and do not distinguish the senders, the systems we consider are symmetric. This allows us to represent a global state — a configuration — by (process) counters: Instead of recording which process is in which local state (which is done typically in distributed algorithms theory), we capture for each local state, how many processes are in it, and then use the rules of the threshold automaton to define the transitions between configurations. In the following, we quickly sketch the semantics to the extent necessary for this paper. Complete definitions can be found in [19].

For every TA we define a counter system as a transition system. First, for every location ℓ we introduce a counter $\boldsymbol{\kappa}[\ell]$ that keeps track of the number of processes in that particular location. A configuration σ is defined as an assignment of all counters of locations, all shared variables from Γ, and all parameters from Π, that respects the resilience condition. If a rule r is an edge (ℓ, ℓ') of a TA, then a transition (r, m) represents m processes moving from the location ℓ to ℓ'. We call m the *acceleration factor.* If $m = 1$ for all transitions in an execution, we get asynchronous executions where one process moves at a time, that is, interleaving semantics. If the rule r has a label $g \mapsto \texttt{act}$, then (r, m) can be applied only in

a configuration in which the counter $\boldsymbol{\kappa}[\ell]$ has a value at least m, and g evaluates to true, and remains true during $m-1$ applications of `act`. In other words, only if the threshold from g is reached, and there are enough processes in location ℓ; see [19] for details. After executing the transition (r, m), counters are updated such that $\boldsymbol{\kappa}[\ell]$ is decreased by m and $\boldsymbol{\kappa}[\ell']$ is increased by m, and shared variables are updated according to the action `act`, m times.

▶ **Example 2.** Consider the TA from Figure 3. One configuration is the following: parameters are $n=7$, $f=t=2$, satisfying resilience condition $n>3t\geq 0 \wedge f\leq t$, counters have values $\boldsymbol{\kappa}[\ell_0]=2$, $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}]=3$, $\boldsymbol{\kappa}[\ell_1]=\boldsymbol{\kappa}[\ell_{\mathsf{AC}}]=0$ and shared variable $\mathsf{echos}=3$. (As we only model correct process explicitly, the counters add up to $n-f=5$.) As in this configuration we have that $\mathsf{echos}+f=5\geq 5=n-t$, and $\boldsymbol{\kappa}[\ell_0]=2$, we can execute transition $(r_2,2)$. The obtained configuration has the same parameter values, but counters are changed: $\boldsymbol{\kappa}[\ell_0]=\boldsymbol{\kappa}[\ell_1]=0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}]=3$, and $\boldsymbol{\kappa}[\ell_{\mathsf{AC}}]=2$. Also, as the action of the rule r_2 is echos++, and two processes are moving along this edge, then the new value of echos is 5. ◁

With a TA we associate a set of predicates $\mathcal{P}_{\mathsf{TA}}$ that track properties of the system states. The set $\mathcal{P}_{\mathsf{TA}}$ consists of the TA's threshold guards and a test $\boldsymbol{\kappa}[\ell]=0$ for every location ℓ in TA. For every configuration σ, one can compute the set $\rho(\sigma)\subseteq\mathcal{P}_{\mathsf{TA}}$ of the predicates that hold true in σ. As was demonstrated in [19], the predicates from $\mathcal{P}_{\mathsf{TA}}$ and linear temporal logic are sufficient to express the safety and liveness properties of threshold-guarded distributed algorithms found in the literature. Essentially, the test $\boldsymbol{\kappa}[\ell]=0$ evaluates to true if no process is in location ℓ, and $\boldsymbol{\kappa}[\ell]\neq 0$ evaluates to true if there is at least one process at ℓ. That all processes are in specific locations can be expressed by a condition that states that "no processes are in the other locations", that is, as a Boolean combination of tests for zero. We include the threshold guards in $\mathcal{P}_{\mathsf{TA}}$ to be able to express the fairness properties such as: if $\mathsf{echos}\geq t+1$, then every process should eventually make one of the transitions labelled with $\mathsf{echos}+f\geq t+1$. Examples of such properties for our benchmarks are given in Section 5.

A system execution is expressed as a path in the counter system. Formally, a *path* is an infinite alternating sequence of configurations and transitions, that is, $\sigma_0, t_1, \sigma_1, \ldots, t_i, \sigma_i, \ldots$, where σ_0 is an initial configuration, and σ_{i+1} is the result of applying t_{i+1} to σ_i for $i\geq 0$. The infinite sequence $\rho(\sigma_0), \rho(\sigma_1), \ldots$ is called the path *trace*. With $\mathsf{Traces}_{\mathsf{TA}}$ we denote the set of all path traces in the TA's counter system. Correctness of a distributed algorithm then means that all traces in $\mathsf{Traces}_{\mathsf{TA}}$ satisfy a specification expressed in linear temporal logic [10]. The verification approach from [19] discussed in Section 3 specifically looks for traces that violate the specification. Such traces are characterized by the temporal logic $\mathsf{ELTL}_{\mathsf{FT}}$ that allows one to express *negations of specifications* relevant for fault-tolerant distributed algorithms.

3 Verification machinery

In [19] we introduced a technique for parameterized verification of threshold-based distributed algorithms. Given a fixed threshold automaton TA, a resilience condition RC, and a set $\{\neg\varphi_1,\ldots,\neg\varphi_k\}$ of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas representing negation of specifications, we check whether there is an execution violating the specification $(\varphi_1\wedge\ldots\wedge\varphi_k)$. Thus, as an output, the algorithm from [19] either confirms correctness, or gives a counterexample. In this paper, we use this technique as a black box, that is, we assume that there is a function

$$\mathsf{verify}_{\mathsf{ByMC}}(\mathsf{TA}, RC, \{\neg\varphi_1,\ldots,\neg\varphi_k\})$$

that either reports a counterexample, or that TA is correct. As our synthesis approach learns from counterexamples, we recall the form of the counterexamples reported by the verifier.

Representative executions and schemas. The idea of [19] lies in automatically computing length and structure of representative executions in advance, whose shape we call schemas. A *schema* is an alternating sequence of contexts (sets of guards) and sequences of rules. Precise definition of a schema and its encoding can be found in [20, 19]. Intuitively, a schema is a concatenation of multiple *simple* schemas. A simple schema has the form $\{g_1, \ldots, g_k\}\, r_1 \ldots r_s\, \{g'_1, \ldots, g'_{k'}\}$, where $k, k', s \in \mathbb{N}$, $g_1, \ldots, g_k, g'_1, \ldots, g'_{k'}$ are guards, and $r_1, \ldots, r_s$ are rules. Given acceleration factors m_i, $1 \le i \le s$, such that $0 \le m_i \le n$, the simple schema generates an execution where all the guards $g_1, \ldots, g_k$ hold in its initial configuration, and after executing $(r_1, m_1), \ldots, (r_s, m_s)$, we arrive in a configuration where all the guards $g'_1, \ldots, g'_{k'}$ hold. As proven in [19], specific schemas of fixed length represent infinite executions (that end in an infinite loop) as required for counterexamples to liveness.

Our verification tool considers each schema in isolation, and basically searches for an evaluation ν of the parameters $(n,\ t,\ f)$, an initial configuration (values of counters of initial local states), and all the acceleration factors, such that the resulting execution is admissible (only enabled rules are executed, etc.). Such an execution — if found — constitutes a counterexample, and the tool reports the corresponding pair (schema, ν).

Solver. Our tool encodes a schema as an SMT formula over parameters, counters of local states, global variables, and acceleration factors. This formula is a conjunction of equalities and inequalities in linear integer arithmetic. Inequalities come from guards, and have shared variables and parameters as free variables. Equalities come from transitions, as every transition is encoded as updating counters of local states and shared variables. The tool ByMC [19] calls an SMT solver to check satisfiability of the formula.

4 Synthesis

Synthesis problem. A temporal logic formula φ in $\mathsf{ELTL}_{\mathsf{FT}}$ describes an (infinite) set of bad traces that the synthesized algorithm must avoid. Therefore, we consider the following formulation of the *synthesis problem*. Given a sketch threshold automaton STA and an (infinite) set of bad traces $\mathsf{Traces}_{\mathsf{Bad}}$, either:

- find an assignment $\mu : U \to \mathbb{Q}$, in order to obtain the fixed threshold automaton $\mathsf{STA}[\mu]$ whose traces $\mathsf{Traces}_{\mathsf{STA}[\mu]}$ *do not intersect* with $\mathsf{Traces}_{\mathsf{Bad}}$, or
- report that no such assignment exists.

Our approach is to find values for the unknowns in a synthesis refinement loop and test them with the verification technique from Section 3.

Synthesis loop. Figure 5 shows the pseudo-code of the synthesis procedure $\mathsf{synt}_{\mathsf{ByMC}}$. At its input the procedure receives a sketch threshold automaton STA, a resilience condition, and a set of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$, which capture the bad traces $\mathsf{Traces}_{\mathsf{Bad}}$. In line 2, formula Θ_0, which captures constraints on the unknowns from U, is initialized using a function $\mathtt{bound}_U$. In principle, $\mathtt{bound}_U$ can be initialized to true (no constraints). However, to ensure termination, we will discuss later in this section, how we obtain constraints that bound the coefficients of sane guards. After initialization we enter the synthesis loop.

The SMT solver checks whether Θ_i has a satisfying assignment to the unknowns in U (line 4). If Θ_i is unsatisfiable, the loop terminates with a *negative outcome* in line 5. Otherwise, the SMT solver gives us an assignment $\mu : U \to \mathbb{Q}$ that is a solution candidate. To check feasibility of μ, the verifier is called for the fixed threshold automaton $\mathsf{STA}[\mu]$ in line 7. The verifier generates multiple schemas, each being one SMT query, which are checked either

```
1  procedure synt_ByMC(STA, RC, {¬φ_1, ..., ¬φ_k})
2    Θ_0 := bound_U(RC) and i := 0
3    while (true)
4      call check_SMT(Θ_i)
5      case unsat ⇒ print 'no more solutions' and exit()
6      case sat(μ) ⇒ /* μ assigns rationals to the variables in U */
7        call verify_ByMC(STA[μ], RC, {¬φ_1, ..., ¬φ_k})
8        case correct ⇒
9          print 'solution μ' /* exclude this solution and continue */
10         Θ_{i+1} := Θ_i ∧ ⋁_{?_j ∈ U} ?_j ≠ μ[?_j] and i := i+1
11       case counterexample(S, ν) ⇒ /* dom(ν) ∩ U = ∅ */
12         S_U := generalize(S, STA)
13         Ψ := formula_SMT(S_U)
14         Θ_{i+1} := Θ_i ∧ ¬Ψ[ν] and i := i+1
```

Figure 5 Pseudo-code of the synthesis loop

sequentially or in parallel. If the verifier reports that a schema that produces a counterexample does not exist, then the candidate assignment μ and threshold automaton $\mathsf{STA}[\mu]$ give us a solution to the synthesis problem. If we were interested in just one solution, the loop would terminate here with a *positive outcome*. However, because we want to enumerate all solutions, our function does a complete search, such that we exclude the solution μ for the future search in line 10, and continue.

If the verifier finds a counterexample, the loop proceeds with the branch in line 11. A counterexample is a schema S of $\mathsf{STA}[\mu]$ and a satisfying assignment $\nu : V \to \mathbb{Q}$ to the free variables V of the SMT formula $\mathsf{formula}_{\mathsf{SMT}}$, which include the parameters Π, shared variables x^j for $x \in \Gamma$, and counters $\boldsymbol{\kappa}^j[\ell]$ for each local state $\ell \in \mathcal{L}$ and every configuration j. In principle, we could exclude μ from consideration similar to line 10. For efficiency, we want to exclude a larger set of evaluations, namely all that lead to the same counterexample: We produce a *generalized* schema S_U, by replacing the rules and guards in S, which belong to the threshold automaton $\mathsf{STA}[\mu]$ with the rules and guards of the sketch threshold automaton STA (line 12). In line 13, we generate a generalized counterexample Ψ. As Ψ is derived from a counterexample with valuations μ and ν, we know that $\Psi[\nu][\mu]$ is true. Further, for every evaluation of the unknowns μ', if $\Psi[\nu][\mu']$ is true, then $\Psi[\nu][\mu']$ is a counterexample. To exclude all these evaluations μ' at once, we conjoin $\neg\Psi[\nu]$ with Θ_i in line 14, which gives us new constraints on the unknowns, before entering the next loop iteration.

The synthesis loop terminates only in line 5, that is, if Θ_i is unsatisfiable. As, in this case, Θ_i is equivalent to false, the following observation guarantees that all satisfying assignments of Θ_0 have been explored and all solutions (if any exists) have been reported.

▶ Observation 1. At the beginning of every iteration $i \geq 0$ of the synthesis loop in lines 3–14, the following invariant holds: if $\mu : U \to \mathbb{Q}$ is a satisfying assignment of formula $\Theta_0 \wedge \neg\Theta_i$, then either: (1) μ was previously reported as a solution in line 9, or (2) μ was previously excluded in line 14 and thus is not a solution. ◁

Completeness and termination for sane guards. Without restricting Θ_0, the search space for coefficients is infinite. In the following, we show that restricting the synthesis problem to sane guards bounds the search space.

The role of threshold guards is typically to check whether the number of distinct senders, from which messages are received, reaches a threshold. We also use threshold guards in our

models to bound the number of processes that go into a special crash state. In both cases, one counts distinct processes and it is therefore natural to consider only those thresholds whose value is in $[0, n]$. More precisely, if the guard has a form $x \geq \bar{u} \cdot \bar{\pi}^{\intercal}$ or $x < \bar{u} \cdot \bar{\pi}^{\intercal}$, then for all parameter values that satisfy resilience condition it holds that $0 \leq \bar{u} \cdot \bar{\pi}^{\intercal} \leq n$. We call such guards *sane* for a given resilience condition.

Theorem 3 considers a general case of hybrid failure models [30] where different failure bounds exist for different failure models (e.g., t_1 Byzantine faults and t_2 crash faults), and these failure bounds are related to the number of processes n by a resilience condition[1] of the form $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$. We bound the values of the coefficients of sane guards.

▶ **Theorem 3.** *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, for $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{or} \quad x < an + (b_1 t_1 + \ldots + b_k t_k) + c,$$

where $x \in \Gamma$, and $a, b_1, \ldots, b_k, c \in \mathbb{Q}$. If the guard is sane for the resilience condition, then

$$0 \leq\ a\ \leq 1, \tag{1}$$

$$-\delta_i - 1 <\ b_i\ < \delta_i + 1, \text{ for all } i = 1, \ldots, k \tag{2}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq\ c\ \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{3}$$

The case when $k = 1$ gives us the classical resilience condition where the system model assumes *one* type of faults (e.g., crash), and the assumed number of faults t is related to the total number of processes n, by a condition $n > \delta t \geq 0$ for some $\delta > 0$. If the guard that compares a shared variable and $an + bt + c$ is sane for the resilience condition, then we obtain that $0 \leq a \leq 1$, $-\delta - 1 < b < \delta + 1$, and $-2\delta - 2 \leq c \leq 2\delta + 2$. Any restriction of the intervals from Theorem 3 to finite sets gives us completeness: If we reduce the domain of variables from U to integers, or to rationals with fixed denominator (e.g., $\frac{z}{10}$ for $z \in \mathbb{Z}$), one reduces the search space to a finite set of valuations. All threshold-based distributed algorithms we are aware of, use guards with coefficients that are either integers or rationals with a denominator not greater than 3. Thus, we restrict our intervals by intersecting them with the set of rational numbers whose denominator is at most D, for a given $D \in \mathbb{N}$.

The following corollary is a direct consequence of Theorem 3, and it tells us how to modify intervals if the coefficients are rational numbers with a fixed denominator.

▶ **Corollary 4.** *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k \right) + \frac{\tilde{c}}{D} \quad \text{or} \quad x < \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k \right) + \frac{\tilde{c}}{D},$$

where $x \in \Gamma$, $\tilde{a}, \tilde{b_1}, \ldots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$, $D \in \mathbb{N}$. If the guard is sane for the resilience condition then

$$0 \leq\ \tilde{a}\ \leq D, \tag{4}$$

$$D(-\delta_i - 1) <\ \tilde{b_i}\ < D(\delta_i + 1), \text{ for all } i = 1, \ldots, k, \tag{5}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k - 1) \leq\ \tilde{c}\ \leq D(2(\delta_1 + \ldots + \delta_k) + k + 1). \tag{6}$$

[1] Because a guard that is sane for a weaker resilience condition, is also sane for a stronger one, Theorem 3 and Corollary 4 also hold for any resilience condition that follows from this one, e.g., $n > \max\{\delta_1 t_1, \ldots, \delta_k t_k\} \wedge \forall i.\ t_i \geq 0$. We can use the same intervals, confirmed by the same proofs as in Appendix A. However, our benchmarks use the form of resilience conditions of Theorem 3.

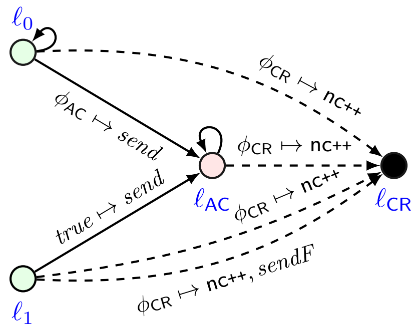


Figure 6 A sketch threshold automaton for folklore reliable broadcast

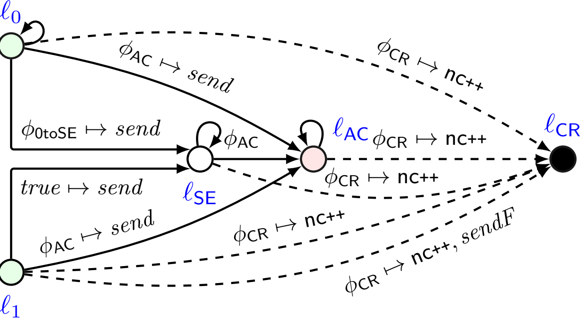


Figure 7 A sketch threshold automaton for reliable broadcast with Byzantine and crash faults

Constraints (4)–(6) constitute the sanity box that function bound_U computes in Figure 5. By fixing D, we restrict Θ_0 to have finitely many satisfying assignments (integers). Hence, the loop terminates. Statements similar to Theorem 3 and Corollary 4 can be derived for other forms of threshold guards, e.g., for thresholds with floor or ceiling functions.[2]

5 Case Studies and Experiments

We have extended ByMC [20, 19] with the synthesis technique presented in this paper. A virtual machine with the tool and the benchmarks is available from: `http://forsyte.at/software/bymc`.[3] ByMC is written in OCaml and uses Z3 [11] as a backend SMT solver. We ran the experiments on two systems: a laptop and the Vienna Scientific Cluster (VSC-3). The laptop is equipped with 16 GB of RAM and Intel® Core™ i5-6300U processor with 4 cores, 2.4 GHz. The cluster VSC-3 consists of 2020 nodes, each equipped with 64 GB of RAM and 2 processors (Intel® Xeon™ E5-2650v2, 2.6 GHz, 8 cores) and is internally connected with an Intel QDR-80 dual-link high-speed InfiniBand fabric: `http://vsc.ac.at`.

We synthesize thresholds for asynchronous fault-tolerant distributed algorithms. We consider *reliable broadcast* and *fast decision* for a consensus algorithm. In the case of reliable broadcast we consider different fault models, namely, crashes [7] and Byzantine faults [29], as well as a hybrid fault model [30] with both, Byzantine and crash failures. For fast decision, we consider the one-step consensus algorithm BOSCO for Byzantine faults [28].

Reliable broadcast for crash and/or Byzantine failures. Figure 7 shows a sketch threshold automaton of a reliable broadcast that should tolerate $f_c \le t_c$ crash and $f_b \le t_b$ Byzantine faults under the resilience condition $n > 3t_b + 2t_c$. For our experiments under simpler failure models — only Byzantine and crash faults — we use the sketch threshold automata from Figures 2 and 6. However, the same thresholds can be obtained by setting $t_c = f_c = 0$ and $t_b = f_b = 0$ in the automaton from Figure 7, respectively. In Figure 2, we do not need a dedicated crash state, as we only model correct processes explicitly, while Byzantine faults are modeled via the guards (cf. Example 1). The automaton from Figure 6 can be obtained from Figure 7 by removing the location ℓ_{SE}.

[2] Theorem 6 and Corollary 7 in Appendix B consider floor and ceiling functions. Our benchmarks do not make use of such thresholds.
[3] See `http://forsyte.at/opodis17-artifact/` for detailed instructions on using the tool.

Table 1 Synthesized solutions for reliable broadcast that tolerates: crashes (Figure 6), Byzantine faults (Figure 2), and Byzantine & crash faults (Figure 7). We used the laptop in the experiments.

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > t_c,\ t_b = 0$	U, C, R	1	*true*	1	12	6
$n > 3t_b,\ t_c = 0$	U, C, R	3	$n - 2t_b$ $t_b + 1$ $t_b + 1$	$n - t_b$ $2t_b + 1$ $n - t_b$	31	16
$n \geq 3t_b,\ t_c = 0$	U, C, R	None	—	—	25	7
$n > 3t_b + 2t_c$	U, C, R	3	$n - 2t_b - 2t_c$ $t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c$ $n - t_b - t_c$	34	50
$n \geq 3t_b + 2t_c$	U, C, R	None	—	—	21	12
$n > 3t_b + t_c$	U, C, R	None	—	—	29	24

The algorithms we consider are the core of broadcasting algorithms, and establish agreement on whether to accept the message by the broadcaster. Similar to Example 1, processes start in locations ℓ_1 and ℓ_0, which capture that the process has received and has not received a message by the broadcaster, respectively. A correctly designed algorithm should satisfy the following properties [29]:

(U) *Unforgeability:* If no correct process starts in ℓ_1, then no correct process ever enters ℓ_{AC}.
(C) *Correctness:* If all correct processes start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.
(R) *Relay:* Whenever a correct process enters ℓ_{AC}, all correct processes eventually enter ℓ_{AC}.

In the following discussion we use Figure 7 as example. We have to sketch the guards ϕ_{CR}, ϕ_{0toSE}, and ϕ_{AC}. At most f_c processes can move to the crashed state ℓ_{CR}. The algorithm designer does not have control over the crashes, and thus we fix the guard ϕ_{CR} to be $\mathsf{nc} < f_c$: The shared variable nc maintains the actual number of crashes (initially zero), which is used only to model crashes and thus cannot be used in guards other than ϕ_{CR}. To properly model that a processes can crash during a "send to all" operation (*non-clean crash*), we introduce two shared variables: the variable echos stores the number of echo messages that are sent by the correct processes (some of them may crash later), and the variable $\mathsf{echosCF}$ stores the number of echo messages that are sent by the correct processes and the faulty processes when crashing. Hence, the action *send* increases both echos and $\mathsf{echosCF}$, whereas the action *sendF* increases only $\mathsf{echosCF}$.

We define the thresholds τ_{0toSE} and τ_{AC} as $(\mathbf{?}^{\mathsf{SE}}_a \cdot n + \mathbf{?}^{\mathsf{SE}}_b \cdot t_b + \mathbf{?}^{\mathsf{SE}}_c \cdot t_c + \mathbf{?}^{\mathsf{SE}}_d)$ and $(\mathbf{?}^{\mathsf{AC}}_a \cdot n + \mathbf{?}^{\mathsf{AC}}_b \cdot t_b + \mathbf{?}^{\mathsf{AC}}_c \cdot t_c + \mathbf{?}^{\mathsf{AC}}_d)$ respectively. Hence, ϕ_{0toSE} and ϕ_{AC} are defined as $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{0toSE}}$ and $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{AC}}$. As discussed in the introduction, we add f_b to $\mathsf{echosCF}$ to reflect that the correct processes may — although do not have to — receive messages from Byzantine processes. For *reliable communication*, we have to enforce:

$$\textit{Every correct process eventually receives at least } \mathsf{echos} \textit{ messages.} \qquad \text{(RelComm)}$$

As threshold automata do not explicitly store the number of received messages, we transform (RelComm) into a fairness constraint, which forces processes to eventually leave a location if the messages by correct processes alone enable a guard of an edge that is outgoing from this location. That is, *there is a time after which the following holds forever:*

$$\boldsymbol{\kappa}[\ell_1] = 0 \wedge (\mathsf{echos} < \tau_{\mathsf{0toSE}} \vee \boldsymbol{\kappa}[\ell_0] = 0) \wedge (\mathsf{echos} < \tau_{\mathsf{AC}} \vee (\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 0)). \quad \text{(Fair)}$$

Table 2 Synthesized solutions for variations of reliable broadcast and specifications (X)–(Z).

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > 3t_b,\ t_c = 0$	X, C, R	None	—	—	15	2
$n > 3t_b + 2,\ t_c = 0$	X, C, R	3	$n - 2t_b$ $t_b + 3$ $t_b + 3$	$n - t_b$ $2t_b + 3$ $n - t_b$	35	12
$n > 3t_b,\ t_c = 0$	Y, C, R	None	—	—	28	6
$n > 4t_b,\ t_c = 0$	Y, C, R	3	$n - 2t_b$ $2t_b + 1$ $2t_b + 1$	$n - t_b$ $3t_b + 1$ $n - t_b$	33	12
$n > 3t_b + 2t_c$	U, Z, R	2	$t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c + 1$	41	31

Table 1 summarizes the experimental results for reliable broadcast, when looking for integer solutions only. The cases $t_b = 0$ and $t_c = 0$ correspond to the algorithms that tolerate only crashes (Figure 6) and only Byzantine faults (Figure 2) respectively. For these cases, we obtained the solutions known from the literature [29, 7] and some variations. Moreover, when the resilience condition is changed from $n > 3t_b$ to $n \geq 3t_b$, our tool reports no solution, which also complies with the literature [29]. In the case of f_c crashes and f_b Byzantine faults, the tool reports three solutions. Moreover, when we tried to relax the resilience condition to $n \geq 3t_b + 2t_c$ and $n > 3t_b + t_c$, the tool reported that there is no solution, as expected.

Variations of the specification. Our logic allows us to easily change the specifications. For instance, we can replace the precondition of unforgeability "if *no* correct process starts in ℓ_1" by giving an upper bound (number or parameter) on correct processes starting in ℓ_1 that still prevents entering ℓ_{AC}, in specifications (X) and (Y). We also changed the precondition of correctnesss "if *all* correct processes start in ℓ_1" in specification (Z):

(X) If *at most two* correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.
(Y) If *at most* t_b correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.
(Z) If *at least* $t_b + t_c + 1$ non-Byzantine processes (correct or crash faulty) start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

Interestingly, we obtain new distributed computing problems that put quantitative conditions on the initial state. These specifications are related to the specifications of condition-based consensus [27]. Our tool automatically generates solutions, or shows their absence in the case resilience conditions are too strong. Table 2 summarizes these results.

Byzantine one-step consensus. Figure 8 shows a sketch threshold automaton of a one-step Byzantine consensus algorithm that should tolerate $f \leq t$ Byzantine faults under the assumption $n > 3t$. It is a formalization of the BOSCO algorithm [28]. The purpose of the algorithm is to quickly reach consensus if (a) $n > 5t$ and $f = 0$, or (b) $n > 7t$. In this encoding, correct processes make a "fast" decision on 0 or 1 by going in the locations ℓ_{D0} and ℓ_{D1}, respectively. When neither (a) nor (b) holds, the processes precompute their votes in the first step and then go to the locations ℓ_{U0} and ℓ_{U1}, from which an *underlying consensus* algorithm is taking over. In this sense, BOSCO can be seen as an asynchronous preprocessing step for general consensus algorithms, and the properties given below contain preconditions

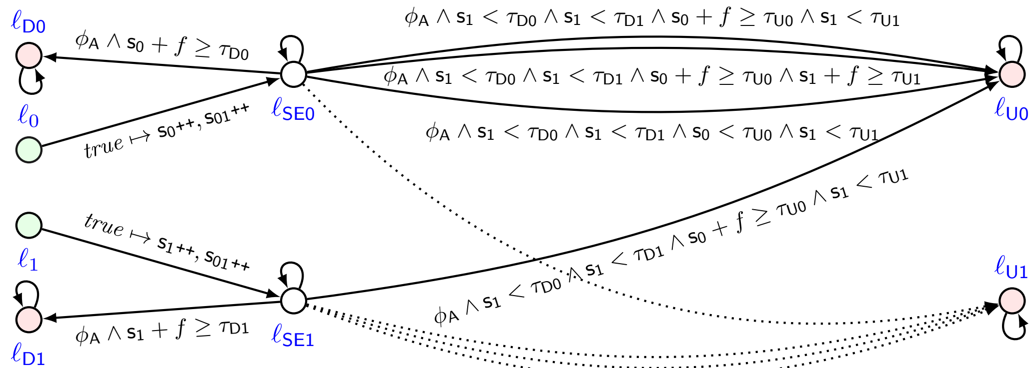


Figure 8 A sketch threshold automaton for one-step Byzantine consensus. Labels of dashed edges are omitted; they can be obtained from the respective solid edges by swapping 0 and 1.

for calling consensus in a safe way (see Fast Agreement below). Every run of a synthesized threshold automaton must satisfy the following properties (for $i \in \{0, 1\}$ and $j = 1 - i$):

(A) *Fast agreement [28, Lemmas 3–4]:* Condition $\boldsymbol{\kappa}[\ell_{Di}] \neq 0$ implies $\boldsymbol{\kappa}[\ell_{Dj}] = \boldsymbol{\kappa}[\ell_{Uj}] = 0$.

(O) *One step:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_j] = 0$, then it always holds that $\boldsymbol{\kappa}[\ell_{Dj}] = 0$ and $\boldsymbol{\kappa}[\ell_{U0}] = \boldsymbol{\kappa}[\ell_{U1}] = 0$. That is, the underlying consensus is never called.

(F) *Fast termination:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_j] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

(T) *Termination:* It eventually holds that $\boldsymbol{\kappa}[\ell_0] = \boldsymbol{\kappa}[\ell_1] = 0$ and $\boldsymbol{\kappa}[\ell_{SE0}] = \boldsymbol{\kappa}[\ell_{SE1}] = 0$.

We define thresholds τ_A, τ_{D0}, τ_{D1}, τ_{U0}, τ_{U1} as $?^x_a \cdot n + ?^x_b \cdot t + ?^x_c$ for $x \in \{A, D0, D1, U0, U1\}$. Then, the guard ϕ_A is defined as: $s_{01} + f \geq \tau_A$. Interestingly, the thresholds appear in different roles in the guards, e.g., $s_0 + f \geq \tau_{D0}$ and $s_0 < \tau_{D0}$. These cases correspond to BOSCO's decisions on how many messages *have been received* and how many messages *have not been received* "modulo Byzantine faults."

As with reliable broadcast, we model reliable communication with the following fairness constraint: For $i \in \{0, 1\}$, *from some point on*, the following holds: $\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0 \wedge (s_{01} < \tau_A \vee s_i < \tau_{Di} \vee \boldsymbol{\kappa}[\ell_{SEi}] = 0)$.

We bound *denominators of rationals with two* and use the sanity box provided by Corollary 4. To reduce the search space, we assume that the guards for 0 and 1 are *symmetric*, that is $?^{D0}_a = ?^{D1}_a$ and $?^{U0}_a = ?^{U1}_a$. Still, BOSCO is a challenging benchmark for verification [19] and synthesis. Since the verification procedure from Section 3 independently checks schemas with SMT, we *parallelized* schema checking with OpenMPI, and ran the experiments at Vienna Scientific Cluster (VSC-3) using 8–128 cores; Table 3 summarizes the results. The tool has found four solutions for the guards: $\tau_A = n - t\ [-\frac{1}{2}]$, $\tau_{D0} = \tau_{D1} = \frac{n+3t+1}{2}$, and $\tau_{U0} = \tau_{U1} = \frac{n-t}{2}\ [+\frac{1}{2}]$. In addition to the guards from [28], the tool also reported that one can add or subtract ½ from several guards. Figure 9 demonstrates that increasing the number of cores above 64 slows down synthesis times for this benchmark.

Variations of the BOSCO specifications. We relaxed the precondition for fast termination:

(U) If $n \geq 5t \wedge f = 0$ and initially $\boldsymbol{\kappa}[\ell_j] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

Specs	Nr. of solutions	Calls to verifier	Nr. of cores	Time min.
AOFT	4	516	128	39
AOFT	4	432	96	25
AOFT	4	425	64	24
AOFT	4	502	16	44
AOFT	4	440	8	51
AOUT	0	376	8	40
AOVT	0	337	8	33

Table 3 Experiments for one-step Byzantine consensus for $n > 3t$ running the parallel verifier at VSC-3

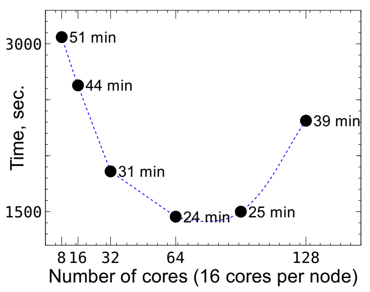


Figure 9 Synthesis times for BOSCO at Vienna Scientific Cluster (VSC-3)

(V) If $n \geq 7t$ and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

As can be seen from Table 3, specifications (U) and (V) have no solutions.

6 Discussions

The classic approach to establish correctness of a distributed algorithm is to start with a system model, a specification, and pseudo code, all given in natural language and mathematical definitions, and then write a manual proof that confirms that "all fits together." Manual correctness proofs mix code inspection, system assumptions, and reasoning about events in the past and the future. Slight modifications to the system assumptions or the code require us to redo the proof. Thus, the proofs often just establish correctness of the algorithm, rather than deriving details of the algorithm — like the threshold guards — from the system assumption or the specification.

We introduced an automated method that synthesizes a correct distributed algorithm from the specifications and the basic assumptions. Our tool computes threshold expressions from the resilience condition and the specification, by learning the constraints that are derived from counterexamples. Learning dramatically reduces the number of verifier calls. In case of BOSCO, the sanity box contains 2^{36} vectors of unknowns, which makes exhaustive search impractical, while our technique only needs to check approximately 500 vectors.

In addition to synthesizing known algorithms from the literature, we considered several modified specifications. For some of them, our tool synthesizes thresholds, while for others it reports that no algorithm of a specific form exists. The latter results are indeed impossibility results (lower bounds on the fraction of correct processes) for fixed sketch threshold automata.

To ensure termination of the synthesis loop, we restrict the search space, and thus the class of algorithms for which the impossibility result formally applies. First, while we restrict the search to sane guards, the same synthesis loop can also be used to synthesize other guards. However, in order to ensure termination, a suitable characterization of sought-after guards should be provided by the user. Second, for reliable broadcast we consider only threshold guards with integer coefficients that can express thresholds like $n - t$ or $2t + 1$. For BOSCO, we only allow division by 2, and can express thresholds like $\frac{n}{2}$ or $\frac{n-t}{2}$. While from a theoretical viewpoint these restrictions limit the scope of our results, we are not aware of a distributed algorithm where processes wait for messages from, say, $\frac{n}{7}$ or $\frac{n}{1000}$ processes. To strengthen our completeness claim, we would need to formally explain why only small denominators are used in fault-tolerant distributed algorithms.

References

1. Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.
2. K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.
3. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.
4. Benjamin Bisping, Paul-David Brodmann, Tim Jungnickel, Christina Rickmann, Henning Seidler, Anke Stüber, Arno Wilhelm-Weidner, Kirstin Peters, and Uwe Nestmann. A constructive proof for FLP. *Archive of Formal Proofs*, 2016.
5. Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and Byzantine-resilient distributed systems. In *CAV*, volume 9779 of *LNCS*, pages 157–176, 2016.
6. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
7. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
8. Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
9. Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, volume 6173 of *LNCS*, pages 142–148, 2010.
10. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
11. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. 2008.
12. Danny Dolev, Keijo Heljanko, Matti Järvisalo, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, and Siert Wieringa. Synchronous counting and computational algorithm design. *J. Comput. Syst. Sci.*, 82(2):310–332, 2016.
13. Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.
14. Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *TAAS*, 10(3):21:1–21:26, 2015.
15. Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *FORTE*, volume 9688 of *LNCS*, pages 124–141, 2016.
16. Adrià Gascón and Ashish Tiwari. A synthesized algorithm for interactive consistency. In *NFM*, volume 8430 of *LNCS*, pages 270–284. Springer, 2014.
17. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.
18. Swen Jacobs and Roderick Bloem. Parameterized synthesis. *LMCS*, 10(1:12), 2014.
19. Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734, 2017.

20 Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

21 Igor Konnov, Josef Widder, Francesco Spegni, and Luca Spalazzi. Accuracy of message counting abstraction in fault-tolerant distributed algorithms. In *VMCAI*, pages 347–366, 2017.

22 Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

23 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370, 2016.

24 Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

25 Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV*, pages 217–237, 2017.

26 Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *SSS*, volume 8756 of *LNCS*, pages 237–251, 2014.

27 Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

28 Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

29 T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

30 Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Dist. Comp.*, 20(2):115–140, 2007.

31 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

APPENDIX

A Detailed Proofs

In order to prove Theorem 3, we first prove mathematical background of it, i.e., Lemma 5.

▶ **Lemma 5.** *Fix a $k \in \mathbb{N}$, and for every $i \in \{1, \dots, k\}$ fix $\delta_i > 0$. Let $a, b_1, \dots, b_k, c$ be rationals for which the following holds: for every $n, t_1, \dots, t_k \in \mathbb{N}$ such that $n > \sum_{i=1}^{k} \delta_i t_i \geq 0$, it holds that $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$. Then it is the case that*

$$0 \leq a \leq 1, \tag{7}$$

$$-\delta_i - 1 < b_i < \delta_i + 1, \text{ for all } i = 1, \dots, k \tag{8}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{9}$$

Proof. Let $\mathbf{P}_{RC}$ be the set of all tuples $(n, t_1, \dots, t_k) \in \mathbb{N}^{k+1}$ that satisfy $n > \sum_{i=1}^{k} \delta_i t_i \geq 0$. Thus, we assume that for $a, b_1, \dots, b_k, c \in \mathbb{Q}$ the following holds:

$$0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n, \text{ for all } (n, t_1, \dots, t_k) \in \mathbf{P}_{RC}. \tag{10}$$

We show that if any of the conditions (7)–(9) is violated, we obtain a contradiction by finding $(n^0, t_1^0, \dots, t_k^0) \in \mathbf{P}_{RC}$ such that $0 \leq an^0 + \sum_{i=1}^{k} b_i t_i^0 + c \leq n^0$ does not hold.

Proof of (7). Let us first show that $0 \le a \le 1$.

Assume by contradiction that $a > 1$. From (10) we know that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $n \ge an + \sum_{i=1}^k b_i t_i + c$, that is, $(1-a)n \ge \sum_{i=1}^k b_i t_i + c$. Since $1 - a < 0$, we obtain

$$n \le \frac{\sum_{i=1}^k b_i t_i + c}{1-a}, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{11}$$

Consider any tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ where $n^0 > \max\left\{\sum_{i=1}^k \delta_i t_i^0, \frac{\sum_{i=1}^k b_i t_i^0 + c}{1-a}\right\}$. By construction, we obtain: (i) the tuple is in $\mathbf{P}_{RC}$ because $n^0 > \sum_{i=1}^k \delta_i t_i^0$, and (ii) we have $n^0 > \frac{\sum_{i=1}^k b_i t_i^0 + c}{1-a}$, such that we arrive at the required contradiction to (11).

Assume now that $a < 0$. Again from (10) we have that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{i=1}^k b_i t_i + c \ge 0$, or in other words $an \ge -\sum_{i=1}^k b_i t_i - c$. As $a < 0$, this means that

$$n \le \frac{-\sum_{i=1}^k b_i t_i - c}{a}, \text{ for every } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{12}$$

Consider a tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ with $n^0 > \max\left\{\sum_{i=1}^k \delta_i t_i^0, \frac{-\sum_{i=1}^k b_i t_i^0 - c}{a}\right\}$. By construction it holds that $n^0 > \sum_{i=1}^k \delta_i t_i^0$, and thus the tuple is in $\mathbf{P}_{RC}$. Also by construction it holds that $n^0 > \frac{-\sum_{i=1}^k b_i t_i^0 - c}{a}$ which is a contradiction with (12).

Proof of (8). Let us now prove that $-\delta_i - 1 < b_i < \delta_i + 1$, for an arbitrary $i \in \{1, \ldots, k\}$.

Assume by contradiction that $b_i \ge \delta_i + 1$. Recall from (10) that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{j=1}^k b_j t_j + c \le n$, or in other words $(1-a)n \ge \sum_{j=1}^k b_j t_j + c$. Since $a \in [0,1]$, then $(1-a)n \le n$, for every $n \ge 0$. Since $b_i \ge \delta_i + 1$, and $t_i \ge 0$, it holds that $b_i t_i \ge (\delta_i + 1)t_i$. Thus, we have that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds that

$$n \ge (1-a)n \ge \sum_{j=1}^k b_j t_j + c \ge (\delta_i + 1)t_i + \sum_{j \ne i} b_j t_j + c.$$

In other words, we have that

$$(n - \delta_i t_i) - \sum_{j \ne i} b_j t_j - c \ge t_i, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{13}$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ such that $t_i^0 = \max\{1, \sum_{j \ne i}(\delta_j - b_j) - c + 2\}$, $t_j^0 = 1$ for $j \ne i$, and $n^0 = \sum_{j=1}^k \delta_j t_j^0 + 1 = \sum_{j \ne i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{j=1}^k \delta_j t_j^0$. Let us check the inequality from (13). By construction we have $(n^0 - \delta_i t_i^0) - \sum_{j \ne i} b_j t_j^0 - c = \sum_{j \ne i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 - \sum_{j \ne i} b_j - c$, that is, $\sum_{j \ne i}(\delta_j - b_j) - c + 1$, which is strictly smaller than t_i^0 by construction. Thus, we obtained a contradiction with (13).

Let us now assume $b_i \le -\delta_i - 1$. Recall from (10) that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $0 \le an + \sum_{j=1}^k b_j t_j + c$. Since $a \in [0,1]$, for every $n \in \mathbb{N}$ holds $an \le n$, and since $b_i \le -\delta_i - 1$, we have $b_i t_i \le -\delta_i t_i - t_i$, for every $t_i \ge 0$. Thus, for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ we have

$$0 \le an + \sum_{j=1}^k b_j t_j + c \le n + (-\delta_i t_i - t_i) + \sum_{j \ne i} b_j t_j + c.$$

In other words, we have that

$$t_i \le (n - \delta_i t_i) + \sum_{j \ne i} b_j t_j + c, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{14}$$

Consider the tuple $(n^0, t_1^0, \dots, t_k^0) \in \mathbb{N}^{k+1}$ where $t_i^0 = \max\{\sum_{j\neq i}(\delta_j + b_j) + c + 2, 1\}$, $t_j^0 = 1$, for every $j \neq i$, and $n^0 = \sum_{j=1}^k \delta_j t_j^0 + 1 = \sum_{j\neq i}\delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$, since $n^0 > \sum_{i=1}^k \delta_i t_i^0$. Let us check the inequality from (14). By construction we have $(n^0 - \delta_i t_i^0) + \sum_{j\neq i} b_j t_j^0 + c = \sum_{j\neq i}\delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 + \sum_{j\neq i} b_j + c$, that is, $\sum_{j\neq i}(\delta_j + b_j) + c + 1$, which is strictly smaller than t_i^0 by construction. This gives us a contradiction with (14).

Proof of (9). And finally, let us prove that $-2\sum_{i=1}^k \delta_i - k - 1 \leq c \leq 2\sum_{i=1}^k \delta_i + k + 1$.

Assume by contradiction that $c > 2\sum_{i=1}^k \delta_i + k + 1$. Recall that for every $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^k b_i t_i + c$, by (10). Since $a \geq 0$, $b_i > -\delta_i - 1$, for every $i = 1, \dots, k$, and $c > 2\sum_{i=1}^k \delta_i + k + 1$, then we have that

$$n \geq an + \sum_{i=1}^k b_i t_i + c > \sum_{i=1}^k (-\delta_i - 1)t_i + 2\sum_{i=1}^k \delta_i + k + 1, \text{ for all } (n, t_1, \dots, t_k) \in \mathbf{P}_{RC}. \quad (15)$$

Consider the tuple $(n^0, t_1^0, \dots, t_k^0)$ where $t_1^0 = \dots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. The tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $\sum_{i=1}^k(-\delta_i - 1)t_i^0 + 2\sum_{i=1}^k \delta_i + k + 1 = \sum_{i=1}^k \delta_i + 1 = n^0$, which is a contradiction with (15).

Assume by contradiction that $c < -2\sum_{i=1}^k \delta_i - k - 1$. Recall that for all $(n, t_1, \dots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{i=1}^k b_i t_i + c$, by (10). Since $a \leq 1$, $b_i < \delta_i + 1$, for every $i = 1, \dots, k$, and $c < -2\sum_{i=1}^k \delta_i - k - 1$, then we have that

$$0 \leq an + \sum_{i=1}^k b_i t_i + c < n + \sum_{i=1}^k (\delta_i + 1)t_i - 2\sum_{i=1}^k \delta_i - k - 1, \text{ for all } (n, t_1, \dots, t_k) \in \mathbf{P}_{RC}. \quad (16)$$

Consider the tuple $(n^0, t_1^0, \dots, t_k^0)$ where $t_1^0 = \dots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $n^0 + \sum_{i=1}^k (\delta_i + 1)t_i^0 - 2\sum_{i=1}^k \delta_i - k - 1 = 0$, which is a contradiction with (16). ◀

Proof of Theorem 3. As the given guard is sane for the resilience condition, the number compared against a shared variable should have a value from 0 to n. For every tuple $(n, t_1, \dots, t_k)$ of parameter values satisfying the resilience condition, it should hold that $0 \leq an + \sum_{i=1}^k b_i t_i + c \leq n$. We may thus apply Lemma 5 and the theorem follows. ◀

Proof of Corollary 4. Using the fact that $x \leq \frac{\tilde{d}}{D} \leq y$ implies that $Dx \leq \tilde{d} \leq Dy$, for a $D \in \mathbb{N}$, this corollary follows directly from Theorem 3. ◀

B Thresholds with floor and ceiling functions

The following theorem considers threshold guards that use the ceiling or the floor function. It uses the same reasoning as in Theorem 3, combined with the properties of these functions. Namely, for every $x \in \mathbb{R}$ it holds that $x \leq \lceil x \rceil < x + 1$ and $x - 1 < \lfloor x \rfloor \leq x$.

▶ **Theorem 6.** *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^k \delta_i t_i \wedge \forall i.t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \dots, k$, and $n, t_1, \dots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(an + (b_1 t_1 + \dots + b_k t_k) + c\right) \quad \text{or} \quad x < f\left(an + (b_1 t_1 + \dots + b_k t_k) + c\right),$$

where $x \in \Gamma$ is a shared variable, $a, b_1, \dots, b_k, c \in \mathbb{Q}$ are rationals, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds that

$$0 \leq a \leq 1, \tag{17}$$
$$-\delta_i - 1 < b_i < \delta_i + 1, \text{ for all } i = 1, \dots, k, \tag{18}$$
$$-2(\delta_1 + \ldots + \delta_k) - k - 2 \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k, \text{ if } f \text{ is floor, or} \tag{19}$$
$$-2(\delta_1 + \ldots + \delta_k) - k \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k + 2, \text{ if } f \text{ is ceiling.} \tag{20}$$

Proof sketch. The proof largely follows the arguments of the proof of Lemma 5 with fixed denominators as in Corollary 4. The only remaining issue is that instead of constraints of the form $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$, that are considered in Lemma 5, here we have to argue about constraints of the form $0 \leq f\left(an + \sum_{i=1}^{k} b_i t_i + c\right) \leq n$, where f is the ceiling or the floor function.

Let us first discuss the case when f is the ceiling function. As for every $x \in \mathbb{R}$ holds that $x \leq \lceil x \rceil < x + 1$, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil < an + (b_1 t_1 + \ldots + b_k t_k) + c + 1.$$

Still, as the guard is sane, we have that $0 \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil \leq n$. Combining these two constraints, we obtain that

$$0 < an + (b_1 t_1 + \ldots + b_k t_k) + (c + 1) \quad \text{and} \quad an + (b_1 t_1 + \ldots + b_k t_k) + c \leq n.$$

With these constraints, we can derive a contradiction following the proof of Lemma 5.

Similarly, if f is the floor function, we use the fact that for every $x \in \mathbb{R}$ holds that $x - 1 < \lfloor x \rfloor \leq x$. Therefore, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c - 1 < \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq an + (b_1 t_1 + \ldots + b_k t_k) + c.$$

As $0 \leq \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq n$, we obtain that

$$0 \leq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{and} \quad an + (b_1 t_1 + \ldots + b_k t_k) + (c - 1) < n.$$

And again, the rest of the proof follows the line of the proof of Lemma 5. ◀

If coefficients in guards have a fixed denominator, we can obtain intervals for numerators as a direct consequence of Theorem 6.

▶ **Corollary 7.** *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \dots, k$, and $n, t_1, \dots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(\frac{\tilde{a}}{D} n + \sum_{i=1}^{k} \frac{\tilde{b_i}}{D} t_i + \frac{\tilde{c}}{D}\right) \quad \text{or} \quad x < f\left(\frac{\tilde{a}}{D} n + \sum_{i=1}^{k} \frac{\tilde{b_i}}{D} t_i + \frac{\tilde{c}}{D}\right),$$

where $x \in \Gamma$ is a shared variable, $\tilde{a}, \tilde{b_1}, \dots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$ are integers, $D \in \mathbb{N}$, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds

$$0 \leq a \leq D, \tag{21}$$
$$D(-\delta_i - 1) < b_i < D(\delta_i + 1), \text{ for all } i = 1, \dots, k, \tag{22}$$
$$D(-2(\delta_1 + \ldots + \delta_k) - k - 2) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k), \text{ if } f \text{ is floor, or} \tag{23}$$
$$D(-2(\delta_1 + \ldots + \delta_k) - k) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k + 2), \text{ if } f \text{ is ceiling.} \tag{24}$$

Part IV

Parameterized Extension of Behavior-Interaction-Priority Framework

Chapter 8

Parameterized Systems in BIP: Design and Model Checking

Igor Konnov, Tomer Kotek, Qiang Wang, Helmut Veith, Simon Bliudze, Joseph Sifakis. Parameterized Systems in BIP: Design and Model Checking. CONCUR, pp. 30:1–30:16, 2016.

DOI: http://dx.doi.org/10.4230/LIPIcs.CONCUR.2016.30

Parameterized Systems in BIP: Design and Model Checking*

Igor Konnov[1], Tomer Kotek[2], Qiang Wang[3], Helmut Veith[4], Simon Bliudze[5], and Joseph Sifakis[6]

1 **TU Wien (Vienna University of Technology), Austria**
`Konnov@forsyte.at`

2 **TU Wien (Vienna University of Technology), Austria**
`Kotek@forsyte.at`

3 **École polytechnique fédérale de Lausanne, Switzerland**
`Qiang.Wang@epfl.ch`

4 **TU Wien (Vienna University of Technology), Austria**
`Veith@forsyte.at`

5 **École polytechnique fédérale de Lausanne, Switzerland**
`Simon.Bliudze@epfl.ch`

6 **École polytechnique fédérale de Lausanne, Switzerland**
`Joseph.Sifakis@epfl.ch`

Abstract

BIP is a component-based framework for system design built on three pillars: behavior, interaction, and priority. In this paper, we introduce first-order interaction logic (FOIL) that extends BIP without priorities to systems parameterized in the number of components. We show that FOIL captures classical parameterized architectures such as token-passing rings, cliques of identical components communicating with rendezvous or broadcast, and client-server systems.

Although the BIP framework includes efficient verification tools for statically-defined systems, none are available for parameterized systems with an unbounded number of components. On the other hand, the parameterized model checking literature contains a wealth of techniques for systems of classical architectures. However, application of these results requires a deep understanding of parameterized model checking techniques and their underlying mathematical models. To overcome these difficulties, we introduce a framework that automatically identifies parameterized model checking techniques applicable to a BIP design. To our knowledge, this is the first framework that allows one to apply prominent parameterized model checking results in a systematic way.

1998 ACM Subject Classification [Software Engineering] D.2.2: Design Tools and Techniques, D.2.4 Software/Program Verification

Keywords and phrases Rigorous system design, BIP, verification, parameterized model checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.30

* We dedicate this article to the memory of Helmut Veith, who passed away tragically while this manuscript was being prepared. His curiosity and energy ignited our joint effort in this research.
This work was supported by the Austrian National Research Network S11403-N23 (RiSE), the Vienna Science and Technology Fund (WWTF) through the grant APALACHE (ICT15-103), and, partially, by the Swiss National Science Foundation through the National Research Programme "Energy Turnaround" (NRP 70) grant 153997.

© Igor Konnov, Tomer Kotek, Qiang Wang, Helmut Veith, Simon Bliudze, and Joseph Sifakis;
licensed under Creative Commons License CC-BY
27th International Conference on Concurrency Theory (CONCUR 2016).
Editors: Josée Desharnais and Radha Jagadeesan; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Design, manufacture and verification of large scale complex hardware/software systems (e.g., cyber-physical systems) remains a grand challenge in system design automation [25]. To address this challenge, the rigorous system design methodology [24] and the behaviour-interaction-priority (BIP) framework [4] have been recently proposed. BIP comes with a formal framework and a toolchain. The BIP framework has well-defined semantics for modeling system behavior and architectures. The BIP toolchain supports verification of high-level system designs and automatic system synthesis of low-level implementations from high-level system designs.

The existing BIP tools focus on design and verification of systems with a fixed number of communicating components [5, 22]. However, many distributed systems are designed with parameterization in mind. For instance, the number of components in the system is not typically fixed, but varies depending on the system setup. In this case, one talks about parameterized verification, where the number of components is a parameter.

Model checking is a pragmatic approach to verification that has found many applications in industry, e.g., see [19]. Many efforts were invested into extension of model checking to the parameterized case, which led to numerous parameterized model checking techniques (see [9] for a recent survey). Unfortunately, often parameterized model checking techniques come with their own mathematical models, which makes their practical application difficult. To perform parameterized model checking, the user has to thoroughly understand the research literature. Typically, the user needs to first manually inspect the parameterized models and match them with the mathematical formalisms from the relevant parameterized verification techniques. Using the match, the user would then apply the decidability results (if any) for the parameterized models, e.g., by computing a cutoff or translating the parameterized model into the language of a particular tool for the specific architecture. Thus, there is a gap between the mathematical formalisms and algorithms from the parameterized verification research and the practice of parameterized verification, which is usually done by engineers who are not familiar with the details of the research literature. In this paper, we aim at closing this gap by introducing a framework for design and verification of parameterized systems in BIP. With this framework, we make the following contributions:

1. We extend propositional interaction logic to the parameterized case with arithmetics, which we call *first-order interaction logic* (FOIL). We build on the ideas from configuration logic [21] and dynamic BIP [10]. FOIL is powerful enough to express architectures found in distributed systems, including the classical architectures: token-passing rings, rendezvous cliques, broadcast cliques, and rendezvous stars. We also identify a decidable fragment of FOIL which has important applications in practice. This contribution is covered by Section 3.
2. We provide a framework for integration of mathematical models from the parameterized model checking literature in an automated way: given a parameterized BIP design, our framework detects parameterized model checking techniques applicable to this design. This automation is achieved by the use of SMT solvers and standard (non-parameterized) model checkers. This contribution is covered by Sections 4 and 5.
3. We provide a preliminary prototype implementation of the proposed framework. Our prototype tool takes a parameterized BIP design as its input and detects whether one of the following classical results applies to this BIP design: the cut-off results for token-passing rings by Emerson & Namjoshi [16], the VASS-based algorithms by German & Sistla [18], and the undecidability and decidability results for broadcast systems by Abdulla et al. [1]

and Esparza et al. [17]. More importantly, our framework is not specifically tailored to the mentioned techniques. This contribution is covered by Sections 5 and 6.

We remark that our framework builds on the notions of BIP, which allows us to express complex notions in terminology understood by engineers. Moreover, our framework allows an expert in parameterized model checking to capture seminal mathematical models found in the verification literature, e.g., [18, 17, 16, 13].

This paper is structured as follows. In Section 2, we briefly recall the BIP modeling framework. In Section 3, we introduce our parameterized extension. In Sections 4 and 5, we present our verification framework and the automatic system architecture identification technique. In Section 6, we present the preliminary experiments. Section 7 closes with related work, conclusions, and future work.

2 BIP without priorities

In this section, we review the notions of BIP [4] with the following restrictions: (i) states of the components do not have specific internal structure; (ii) we do not consider interaction priorities. While we believe that our approach can be extended to priorities, we leave this for future work.

As usual, a labeled transition system is a tuple (S, s_0, A, R) with a set of locations S, an initial location $s_0 \in S$, a non-empty set of actions A, and a transition relation $R \subseteq S \times A \times S$.

▶ **Definition 2.1** (Component type). A component type is a transition system $\mathbb{B} = \langle \mathbb{Q}, \ell^0, \mathbb{P}, \mathbb{E} \rangle$ over the finite sets $\mathbb{Q}$ and $\mathbb{P}$. By convention, the set of actions $\mathbb{P}$ is called the set of ports.

Ports form the interface of a component type. We assume that, for each location, no two outgoing transitions from this location are labeled with the same port. We also assume that the ports of each component type, as well as the locations, are disjoint.

Let $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle$ be a tuple of component types, where each $\mathbb{B}_i$ is $\langle \mathbb{Q}_i, \ell_i^0, \mathbb{P}_i, \mathbb{E}_i \rangle$ for $i \in [0, k)$. We introduce an infinite set of components $\{\mathbb{B}_i[j] \mid j \geq 0\}$ for $i \in [0, k)$. A *component* $\mathbb{B}_i[j] = \langle \mathbb{Q}_i[j], \ell_i^0[j], \mathbb{P}_i[j], \mathbb{E}_i[j] \rangle$ is obtained from the component type $\mathbb{B}_i$ by renaming the set of ports. Thus, as transition systems, $\mathbb{B}_i[j]$ and $\mathbb{B}_i$ are isomorphic. We postulate $\mathbb{P}_i[j] \cap \mathbb{P}_i[j'] = \emptyset$, for $j \neq j'$.

A BIP model is a composition of finitely many components instantiated from the component types $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle$. To denote the number of components of each type, we introduce a size vector $\bar{N} = \langle N_0, \dots, N_{k-1} \rangle$: there are N_i components of component type $\mathbb{B}_i$, for $i \in [0, k)$.

Coordination of components is specified with interactions. Intuitively, an interaction defines a multi-party synchronization of component transitions. A BIP interaction is a finite set of ports, which defines a possible synchronization among components.

▶ **Definition 2.2** (Interaction). Given a tuple of component types $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle$ and a size vector $\bar{N} = \langle N_0, \dots, N_{k-1} \rangle$, an interaction $\gamma \subseteq \{p \in \mathbb{P}_i[j] \mid i \in [0, k), j \in [0, N_i)\}$ is a set of ports such that $|\gamma \cap \mathbb{P}_i[j]| \leq 1$ for all $i \in [0, k)$ and $j \in [0, N_i)$, i.e., an interaction is a set of ports such that at most one port of each component takes part in an interaction. If $p \in \gamma$, we say that p is *active* in γ.

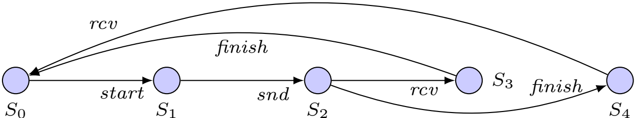
▶ **Definition 2.3** (BIP Model). Given a tuple of component types $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle$ and a size vector $\bar{N} = \langle N_0, \dots, N_{k-1} \rangle$, a BIP model $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle^{\bar{N}, \Gamma}$ is a tuple $\langle \mathcal{B}, \Gamma \rangle$, where $\mathcal{B}$ is the set $\{\mathbb{B}_i[j] \mid i \in [0, k), j \in [0, N_i)\}$ and Γ is a set of interactions defined w.r.t. $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle$ and $\bar{N}$.

▶ **Definition 2.4** (BIP operational semantics). Given a BIP model $\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1}\rangle^{\bar{N},\Gamma}$, we define its operational semantics as a transition system $TS(\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1}\rangle^{\bar{N},\Gamma}) = \langle S, s_0, \Gamma, R\rangle$, where:

1. The set of *configurations* S is defined as the Cartesian product of the sets of locations of the components $\mathbb{Q}_0^{N_0} \times \cdots \times \mathbb{Q}_{k-1}^{N_{k-1}}$. Given a configuration $s \in S$, we denote by $s(i,j)$ the jth member of the tuple defined by the ith product $\mathbb{Q}_i^{N_i}$ where $j \in [0, N_i)$.
2. The initial configuration $s_0 \in S$ satisfies that $s_0(i,j) = \ell_i^0[j]$ for all $i \in [0,k)$ and $j \in [0, N_i)$.
3. The transition relation R contains a triple (s, γ, s'), if, for each $i \in [0,k)$ and $j \in [0, N_i)$, the jth component of type i
   - either has an active port $p \in \gamma \cap \mathbb{P}_i[j]$ and $\langle s(i,j), p, s'(i,j)\rangle \in \mathbb{E}_i[j]$,
   - or is not participating in the interaction γ, i.e., $\gamma \cap \mathbb{P}_i[j] = \emptyset$ and $s'(i,j) = s(i,j)$.

Intuitively, the local transitions of components fire simultaneously, provided that their ports are included in the interaction; other components do not move.

▶ **Example 2.5** (Milner's scheduler). We follow the formulation by Emerson & Namjoshi [16]. A scheduler is modeled as a token-passing ring. Only the process that owns the token may start running a new task. The component type $\mathbb{B}_0 = \langle \mathbb{Q}_0, \ell_0^0, \mathbb{P}_0, \mathbb{E}_0\rangle$ is given by the locations $\mathbb{Q}_0 = \{S_0, \ldots, S_4\}$, the initial location $\ell_0^0 = S_0$, the port types $\mathbb{P}_0 = \{\text{snd}, \text{rcv}, \text{start}, \text{finish}\}$, and the edges $\mathbb{E}_0$ that are shown in the figure below:



A component owns the token when in the location S_0, S_1, or S_3. In S_0, a component initiates its task by interacting on port *start*. The token is then sent to the component's right neighbor on the ring via an interaction on port *snd*. The component then waits until (a) its initiated task has finished, and (b) the component has received the token again. When both (a) and (b) have occurred, the component may initiate a new task. Note that (a) and (b) may occur in either order.

Fix a number $N_0 \in \mathbb{N}$. The following set of interactions represents the ring structure:

$$\Gamma = \{\gamma_{i\to j}, \gamma_{\text{start}(i)}, \gamma_{\text{finish}(i)} \mid 0 \le i < N_0 \text{ and } j \equiv i+1 \bmod n_0\}$$

where $\gamma_{i\to j} = \{(\text{snd}, i), (\text{rcv}, j)\}$ is the interaction passing the token from the ith component to the next component on the ring, while the interactions $\gamma_{\text{start}(i)} = \{(\text{start}, i)\}$ and $\gamma_{\text{finish}(i)} = \{(\text{finish}, i)\}$ allow the ith component to take the internal transitions labeled 'start' and 'finish' respectively. The BIP model of the Milner scheduler of size N_0 is $\langle \mathcal{B}, \Gamma\rangle$, where $\mathcal{B}$ is the set of components $\{\mathbb{B}_0[j] \mid j \in [0, N_0)\}$.

3 Parameterized BIP without priorities

Since the number of possible interactions in a parameterized system is unbounded, and each interaction itself may involve an unbounded number of actions, the set of all possible interactions is infinite. Hence, we need a symbolic representation of such a set. To this end, we propose *first order interaction logic*—a uniform and formal language for system topologies and coordination mechanisms in parameterized systems. Using this logic, we introduce a parameterized extension of BIP, and show that this extension naturally captures standard examples.

3.1 FOIL: First order interaction logic

In this section, we fix a tuple of component types $\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1} \rangle$. For each port $p \in \mathbb{P}_i$ of an ith component type, we introduce a unary *port predicate* with the same name p. Furthermore, we introduce a tuple of constants $\bar{n} = \langle n_0, \ldots, n_{k-1} \rangle$, which represents the number of components of each type. We also assume the standard vocabulary of Presburger arithmetic, that is, $\langle 0, 1, \leq, + \rangle$.

FOIL syntax. Assume an infinite set of index variables $\mathcal{I}$. We say that ψ is a first order interaction logic formula, if it is constructed according to the following grammar:

$$\psi ::= p(i) \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \exists i :: type_j : \phi.\ \psi \mid \forall i :: type_j : \phi.\ \psi\,,$$

where $p \in \mathbb{P}_0 \cup \cdots \cup \mathbb{P}_{k-1}$, $i \in \mathcal{I}$, and ϕ is a formula in Presburger arithmetic over index variables and the vocabulary $\langle 0, 1, \leq, +, \bar{n} \rangle$.

Informally, the syntax $\mathbf{Q}\ i :: type_j : \phi.\ \psi$, where $\mathbf{Q} \in \{\exists, \forall\}$, restricts the index variable i to be associated with the component type $\mathbb{B}_j$. Notice, however, that this syntax does not enforce type correctness of ports. For instance, one can write a formula $\exists i :: type_j : p(i)$ with some $p \notin \mathbb{P}_j$. While this formula is syntactically correct, it is not in line with Definition 2.2 of interaction given in Section 2. To this end, we say that a FOIL formula is *natural*, if for each of its subformulae $\mathbf{Q}\ i :: type_j : \phi.\ \psi(i)$, for $\mathbf{Q} \in \{\exists, \forall\}$, and every atomic formula $p(i)$ of ψ, it holds that $p \in \mathbb{P}_j$. From here on, we assume FOIL formulae to be natural. We write $\exists i :: type_j.\ \psi$ as a shorthand for $\exists i :: type_j : true.\ \psi$.

FOIL semantics. We give the semantics of a FOIL formula by means of structures. A *first-order interaction logic structure* (FOIL structure) is a pair $\xi = (\mathbb{N}, \alpha_\xi)$: the set of natural numbers $\mathbb{N}$ is the domain of ξ, while α_ξ is the interpretation of all the predicates and of the constants $\bar{n}$. The symbols 0, 1, $\leq$, and $+$ have the natural interpretations over $\mathbb{N}$.

A valuation σ is a function $\sigma : \mathcal{I} \to \mathbb{N}$. We denote by $\sigma[x \mapsto j]$ the valuation obtained from σ by mapping the index variable x to the value j. Assignments are used to give values to free variables in formulae. For a FOIL structure ξ and a valuation σ, the semantics of FOIL is formally given as follows (the semantics of Boolean operators and universal quantifiers is defined in the standard way):

$$\begin{array}{lll} \xi, \sigma \models_{\text{FOIL}} p(i) & \textit{iff} & \alpha_\xi(p) \text{ is true on } \sigma(i) \\ \xi, \sigma \models_{\text{FOIL}} \exists i :: type_j : \phi.\ \psi & \textit{iff} & \text{there is } l \in [0, \alpha_\xi(n_j)) \text{ such that} \\ & & \xi, \sigma[i \mapsto l] \models_{\text{FO}} \phi \text{ and } \xi, \sigma[i \mapsto l] \models_{\text{FOIL}} \psi \end{array}$$

where $\models_{\text{FO}}$ to denotes the standard 'models' relation of first-order logic.

Finally, for a FOIL formula ψ without free variables and a structure ξ, we write $\xi \models_{\text{FOIL}} \psi$, if $\xi, \sigma_0 \models_{\text{FOIL}} \psi$ for the valuation σ_0 that assigns 0 to every index $i \in \mathcal{I}$.[1]

Decidability. It is easy to show that checking validity of a FOIL sentence[2] is undecidable, and that FOIL contains an important decidable fragment:

▶ **Theorem 3.1** (Decidability of FOIL). *The following results about FOIL hold:*

[1] Since ψ has no free variables, our choice of σ_0 is arbitrary: for all σ we have $\xi, \sigma \models_{\text{FOIL}} \psi$ if and only if $\xi, \sigma_0 \models_{\text{FOIL}} \psi$.

[2] A FOIL formula with no free variables is called a *sentence*. A sentence is *valid* if it is satisfied by all structures.

(i) *Validity of FOIL sentences is undecidable.*
(ii) *Validity of FOIL sentences in which all additions are of the form $i+1$ is decidable.*

Proof. (i) FOIL contains Presburger arithmetic with unary predicates, which is known to be as strong as Peano arithmetic [20]. Hence, satisfiability and validity of FOIL formulae are undecidable.

(ii) The formula $j = i+1$ is definable in FOIL by $i \leq j \wedge j \neq i \wedge \psi_{consecutive}(i,j)$, where $\psi_{consecutive}(i,j) = \forall \ell :: type_t.\ (j \leq \ell \wedge \ell \leq i) \rightarrow (\ell = i \vee \ell = j)$, where t is the type of i and j. Hence, we can rewrite any FOIL sentence ψ in which all additions are of the form $i+1$ as an equi-satisfiable first-order logic sentence ψ' without using addition (+). The sentence ψ' belongs to S1S, the monadic second order theory of $(\mathbb{N}, 0, 1, \leq)$, which is decidable, see [27]. ◀

In the following, we restrict addition to the form $i+1$, and thus stay in the decidable fragment.

3.2 Interactions as FOIL structures

In contrast to Definition 2.2 of a standard interaction, which is represented explicitly as a finite set of ports, we use first order interaction logic formulae to define all the possible interactions in parameterized systems. Our key insight is that each structure of a formula uniquely defines at most one interaction, and the set of all possible interactions is the union of the interactions derived from the structures that satisfy the formula.

Intuitively, if $p(j)$ evaluates to true in a structure ξ, then the jth instance of the respective component type—uniquely identified by the port p—takes part in the interaction identified with ξ. Thus, we can reconstruct a standard BIP interaction from a FOIL structure by taking the set of ports, whose indices are evaluated to true by the unary predicates. Formally, given a FOIL structure $\xi = (\mathbb{N}, \alpha_\xi)$, we define the set $\gamma_\xi = \{(p,j) \mid i \in [0,k),\ p \in \mathbb{P}_i,\ j \in [0, \alpha_\xi(n_j)),\ \alpha_\xi(p)(j) = \mathit{true}\}$. In the following, the notation (p,j) denotes the port p of the jth component of the type $\mathbb{B}_i$ with $p \in \mathbb{P}_i$.

Notice that γ_ξ does not have to be an interaction in the sense of Definition 2.2. Indeed, one can define ξ whose set γ_ξ includes two ports of the same component. We say that ξ *induces an interaction*, if γ_ξ is an interaction in the sense of Definition 2.2.

▶ **Definition 3.2** (Parameterized BIP Model). A parameterized BIP model is a tuple $\langle \mathbf{B}, \bar{n}, \psi, \epsilon \rangle$, where $\mathbf{B} = \langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1} \rangle$ is a tuple of component types, ψ is a sentence in FOIL over port predicates and a tuple $\bar{n} = \langle n_0, \ldots, n_{k-1} \rangle$ of size parameters, and ϵ is a linear constraint over $\bar{n}$.

The tuple $\bar{n}$ consists of the size parameters for all component types, and the constraint ϵ restricts these parameters. For example, the formula $(n_0 = 1) \wedge (n_1 \geq 10)$ requires every instance of a parameterized BIP model to have only one component of the first type and at least ten components of the second type. The FOIL sentence ψ restricts both the system topology and the communication mechanisms, see Example 3.4.

▶ **Definition 3.3** (PBIP Instance). Given a parameterized BIP model $\langle \mathbf{B}, \bar{n}, \psi, \epsilon \rangle$ and a size vector $\bar{N}$, a *PBIP instance* is a BIP model $\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1} \rangle^{\bar{N}, \Gamma} = \langle \mathcal{B}, \Gamma \rangle$, where $\mathcal{B}$ and Γ are defined as follows:

1. the numbers $\bar{N}$ satisfy the size constraint ϵ,
2. the set of components $\mathcal{B}$ is $\{\mathbb{B}_i[j] \mid i \in [0,k) \text{ and } j \in [0, N_j)\}$, and

3. the set of interactions Γ consists of all interactions γ_ξ induced by a FOIL structure ξ such that the size parameters $\bar{n}$ are interpreted in ξ as $\bar{N}$, and ξ satisfies ψ, i.e. $\alpha_\xi(\bar{n}) = \bar{N}$ and $\xi \models_{\text{FOIL}} \psi$.

In the rest of this section, we give three examples that show expressiveness of parameterized BIP.

▶ **Example 3.4** (Milner's scheduler revisited). The parameterized BIP model of Milner's scheduler is $\langle\langle\mathbb{B}_0\rangle, \langle n_0\rangle, \psi, \mathit{true}\rangle$, where $\mathbb{B}_0$ is from Example 2.5 and $\psi = \psi_{token} \vee \psi_{internal}$ defined as follows. The formula ψ_{token} defines the token-passing interactions and the formula $\psi_{internal}$ defines the internal interactions of starting or finishing a task:

$$\begin{array}{lcl} \psi_{token} & = & \exists i, j :: type_0 : j = (i+1) \bmod n_0.\ snd(i) \wedge rcv(j) \wedge \psi_{only}(i,j) \\ \psi_{only}(i,j) & = & \forall \ell :: type_0 : \ell \neq i \wedge \ell \neq j.\ \neg snd(\ell) \wedge \neg rcv(\ell) \wedge \neg start(i) \wedge \neg finish(i) \\ \psi_{internal} & = & \exists i :: type_0.\ \psi_{only}(i,i) \wedge (start(i) \vee finish(i)) \end{array}$$

The formula ψ_{token} does not have free variables and holds for a structure ξ, if the induced interaction γ_ξ is a send-receive interaction along some edge $i \to j$ of the ring, where $j = (i+1) \bmod n_0$. In fact, $j = (i+1) \bmod n_0$ is just a shorthand for the formula: $(i+1 < n_0 \wedge j = i+1) \vee (i+1 = n_0 \wedge j = 0)$. The formula $\psi_{only}(i,j)$ excludes any component other than i and j from participating in the interaction. (If $i = j$ then all components other than i are excluded.) The formula $\psi_{internal}$ enables the transitions labeled with 'start' and 'finish', in which only one component changes its location.

Observe that the semantics of FOIL forces the quantified variables i, j, ℓ to be in the range from 0 to $N_0 - 1$. Hence, we omit explicit range constraints. For instance, ψ_{token} is equivalent to the formula:

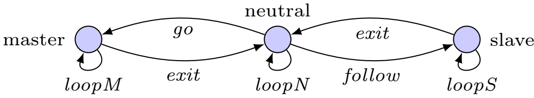
$$\exists i, j :: type_0 : 0 \leq i, j < n_0 \wedge (j = (i+1) \bmod n_0).\ snd(i) \wedge rcv(j) \wedge \psi_{only}(i,j)$$

The set of FOIL structures ξ that satisfy ψ induces the same set of interactions Γ as in Example 2.5. While Example 2.5 defines the set Γ explicitly for any fixed value N_0, in the parameterized setting the interactions are defined uniformly by a single FOIL formula ψ, for all values of N_0.

In this example we do not restrict the initial locations so that exactly one process owns the token in the initial configuration. This delicate issue is resolved in Section 5.4.

▶ **Example 3.5** (Broadcast in a star). Let $\langle\langle\mathbb{B}_0, \mathbb{B}_1\rangle, \langle n_0, n_1\rangle, \psi, \epsilon\rangle$ be a parameterized BIP model with two component types and the size constraint $\epsilon \equiv (n_0 = 1)$. We also assume that component type $\mathbb{B}_0$ (resp. $\mathbb{B}_1$) has only one port *send* (resp. *receive*), i.e., $\mathbb{P}_0 = \{send\}$ and $\mathbb{P}_1 = \{receive\}$. The FOIL formula $\psi = \exists i :: type_0.\ send(i)$ specifies broadcast from the component $\mathbb{B}_0[0]$, the center of the star, to the leaves of type $\mathbb{B}_1$. The set of interactions defined by ψ consists of all sets of ports of the form $\{(send, 0)\} \cup \{(receive, d) \mid d \in D)\}$ for all $D \subseteq [0, n_1)$, including the empty set $D = \emptyset$.

▶ **Example 3.6** (Barrier). Consider a barrier synchronization protocol, cf. [9, Example 6.6]. The component type $\mathbb{B}_0$ is as shown below:



The location *neutral* is the initial location. A synchronization episode consists of three stages:

(i) First, a single component enters the barrier by moving to *master*.
(ii) Then, each of the others components moves to *slave*.
(iii) Finally, the master triggers a broadcast and all components leave the barrier by moving to *neutral*.

The parameterized BIP model of the barrier synchronization protocol is $\langle\langle \mathbb{B}_0\rangle, \langle n_0\rangle, \psi, true\rangle$, where $\psi = \psi_{go} \vee \psi_{follow} \vee \psi_{exit}$, and the following formulae ψ_{go}, ψ_{follow}, and ψ_{exit} describe the interactions of stages (i), (ii), and (iii) respectively:

$$
\begin{aligned}
\psi_{go} &= \exists i :: type_0.\ go(i) \quad \wedge \forall j :: type_0 : i \neq j.\ loopN(j) \\
\psi_{follow} &= \exists i, j :: type_0.\ follow(i) \wedge loopM(j) \wedge \\
&\quad\ \forall \ell :: type_0 : i \neq \ell.\ loopM(\ell) \vee loopN(\ell) \vee loopS(\ell) \\
\psi_{exit} &= \forall i :: type_0.\ exit(i)
\end{aligned}
$$

All three formulae enforce progress by requiring at least one process to change its state.

4 Parameterized model checking

In this section, we review the syntax and semantics of the indexed version of CTL^*, called $ICTL^\star$, which is often used to specify the properties of parameterized systems [9]. Though we use indexed temporal logics to define the standard parameterized model checking problem, these logics are not the focus of this paper. Further, we introduce the parameterized model checking problem for parameterized BIP design, and show its undecidability.

Syntax. For a set of index variables $\mathcal{I}$, the $ICTL^\star$ *state* and *path formulae* follow the grammar:

$$\theta ::= true \mid at(q,i) \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \exists i :: type_j : \phi.\ \theta \mid \forall i :: type_j : \phi.\ \theta \mid \mathbf{E}\,\varphi \mid \mathbf{A}\,\varphi\,, \qquad (state\ formulae)$$

$$\varphi ::= \theta \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\,\varphi \mid \mathbf{F}\,\varphi \mid \mathbf{G}\,\varphi \mid \varphi_1 \mathbf{U}\,\varphi_2\,. \qquad (path\ formulae)$$

where $q \in \bigcup_{0\le j<k} \mathbb{Q}_j$ is a location, $i \in \mathcal{I}$ is an index, and ϕ is a formula in Presburger arithmetic over size variables $\bar{n}$ and index variables from the set $\mathcal{I}$.

Semantics. Fix a BIP model $\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1}\rangle^{\bar{N},\Gamma}$ and its transition system $M = \langle S, s_0, \Gamma, R\rangle = TS(\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1}\rangle^{\bar{N},\Gamma})$ as per Definition 2.4. To evaluate Presburger formulae, we use the first-order structure $PA = \langle \mathbb{N}, 0, 1, \le, +, \bar{N}\rangle$. The semantics of $ICTL^\star$ formulae is defined inductively using M and PA. We only briefly discuss semantics to highlight the role of quantifiers in indexed temporal logics. For further discussions, we refer the reader to the textbook [12].

State formulae are interpreted over a configuration s and a valuation of index variables $\sigma : \mathcal{I} \to \mathbb{N}$ (the semantics of Boolean operators and universal quantifiers is defined in the standard way):

$$
\begin{aligned}
&M, s, \sigma \models_{\mathrm{ICTL}^\star} at(q,i) && \textit{iff} \quad q = s(j, \sigma(i)),\ \text{where } q \in \mathbb{Q}_j \\
&M, s, \sigma \models_{\mathrm{ICTL}^\star} \exists i :: type_j : \phi.\ \theta && \textit{iff} \quad PA, \sigma[i \mapsto l] \models_{\mathrm{FO}} \phi \text{ and } M, s, \sigma[i \mapsto l] \models_{\mathrm{ICTL}^\star} \theta \text{ hold,} \\
& && \qquad \text{for some } l \in [0, N_j) \\
&M, s, \sigma \models_{\mathrm{ICTL}^\star} \mathbf{E}\,\varphi && \textit{iff} \quad M, \pi, \sigma \models_{\mathrm{ICTL}^\star} \varphi \text{ for some infinite path } \pi \text{ starting from } s
\end{aligned}
$$

Path formulae are interpreted over an infinite path π, and the valuation function σ as follows (the semantics for Boolean operators and temporal operators **F** and **G** is defined in the standard way):

$$
\begin{aligned}
&M, \pi, \sigma \models_{\mathrm{ICTL}^\star} \theta && \textit{iff} \quad M, s, \sigma \models_{\mathrm{ICTL}^\star} \theta,\ \text{where } s \text{ is the first configuration of the path } \pi \\
&M, \pi, \sigma \models_{\mathrm{ICTL}^\star} \mathbf{X}\,\varphi && \textit{iff} \quad M, \pi^1, \sigma \models_{\mathrm{ICTL}^\star} \varphi \\
&M, \pi, \sigma \models_{\mathrm{ICTL}^\star} \varphi_1 \mathbf{U}\,\varphi_2 && \textit{iff} \quad \exists j \ge 0.\ M, \pi^j, \sigma \models_{\mathrm{ICTL}^\star} \varphi_2 \text{ and } \forall i < j.\ M, \pi^i, \sigma \models_{\mathrm{ICTL}^\star} \varphi_1,
\end{aligned}
$$

where π^i is the suffix of the path π starting with the ith configuration.

Finally, given a formula φ without free variables, we say that M satisfies φ, written as $M \models_{\text{ICTL}^\star} \varphi$, if $M, s_0, \sigma_0 \models_{\text{ICTL}^\star} \varphi$ for the valuation σ_0 that assigns zero to each index from the set $\mathcal{I}$. The choice of σ_0 is arbitrary, as for all σ, it holds that $M, s_0, \sigma \models_{\text{ICTL}^\star} \varphi$ if and only if $M, s_0, \sigma_0 \models_{\text{ICTL}^\star} \varphi$.

Now we are in the position to formulate the parameterized model checking problem for BIP:

▶ **Problem 4.1** (Parameterized model checking)**.** *The verification problem for a parameterized* BIP *model* $\langle \mathbf{B}, \bar{n}, \psi, \epsilon \rangle$ *and an ICTL** *state formula* θ *without free variables, is whether every instance* $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle^{\bar{N}, \Gamma}$ *satisfies* θ*.*

Not surprisingly, Problem 4.1 is undecidable in general. For instance, one can use the proof idea [16] to obtain the following theorem. We do not give a detailed proof here: to a large extent, it repeats the encoding of a unidirectional token ring, which we discuss later in Section 5.4.

▶ **Theorem 4.2** (Undecidability)**.** *Given a two-counter machine* M_2*, one can construct an ICTL***-formula* $\mathbf{G}\,\neg\mathtt{halt}$ *and a parameterized BIP model* $\mathcal{B} = \langle \mathbf{B}, \bar{n}, \psi, \epsilon \rangle$ *that simulates* M_2 *and has the property:* M_2 *does not halt if and only if* $\langle \mathbb{B}_0, \dots, \mathbb{B}_{k-1} \rangle^{\bar{N}, \Gamma} \models \mathbf{G}\,\neg\mathtt{halt}$ *for all instances of* $\mathcal{B}$*.*

5 Identifying the architecture of a parameterized BIP model

In the non-parameterized case, knowing the architecture is not crucial, as there are model checking algorithms that apply in general to arbitrary finite transition systems. However, the architecture dramatically affects decidability of *parameterized* model checking. Architecture identification plays an important step in our verification framework. In this section, we show how to identify system architectures automatically, and present applications to verification.

Our framework. For the sake of exposition, we assume that parameterized BIP models have only one component type. Our identification framework extends easily to the general case.

Given an architecture $\mathcal{A}$, e.g., the token ring architecture, an expert in parameterized model checking creates formula templates in FOIL (*FOIL-templates*) and in temporal logic (*TL-templates*). *FOIL-templates* describe the system topology and communication mechanism for the architecture $\mathcal{A}$. *TL-templates* describe the behaviour of the component type required by the architecture $\mathcal{A}$, e.g., in a token ring, a component which does not have the token cannot send the token. These templates are designed once for all parameterized BIP models compliant with $\mathcal{A}$. In the sequel, *TL-templates* are only used for token rings, thus we omit them from the discussion of other architectures.

Given a parameterized BIP model $\langle\langle \mathbb{B} \rangle, \langle n \rangle, \psi, \epsilon\rangle$—not necessarily compliant with the architecture $\mathcal{A}$—the templates for the architecture $\mathcal{A}$ are instantiated to FOIL formulae $\varphi_1^{\mathit{FOIL}}, \dots, \varphi_m^{\mathit{FOIL}}$, and temporal logic formulae $\varphi_1^{\mathit{TL}}, \dots, \varphi_\ell^{\mathit{TL}}$. The FOIL formulae guarantee that the set of interactions expressed by the FOIL formula ψ adheres to $\mathcal{A}$. The temporal logic formulae guarantee that the behaviour of the component type $\mathbb{B}$ adheres to $\mathcal{A}$. The *identification criterion* is as follows: if $\varphi_1^{\mathit{FOIL}} \wedge \dots \wedge \varphi_m^{\mathit{FOIL}}$ is valid and $\mathbb{B} \models_{\text{TL}} \varphi_1^{\mathit{TL}} \wedge \dots \wedge \varphi_\ell^{\mathit{TL}}$ holds, then the parameterized model $\langle\langle \mathbb{B} \rangle, \langle n \rangle, \psi, \epsilon\rangle$ is compliant with the architecture $\mathcal{A}$. In practice, we use an SMT solver to check validity of the FOIL formulae and a model checker to check that the component type $\mathbb{B}$ satisfies the temporal formulae.

In the rest of this section we construct FOIL-templates and TL-templates for well-known architectures: cliques of processes communicating via broadcast, cliques of processes communicating via rendezvous, token rings, and server-client systems in which processes are organized in a star and communicate via rendezvous. We show that the provided templates identify the architectures in a sound way.

5.1 The common templates for BIP semantics

As we discussed in Section 3.2, not every FOIL structure induces a BIP interaction. We show that one can write a FOIL-template that restricts FOIL structures to induce BIP interactions. The following template $\eta^{FOIL}_{interaction}(\mathbb{P}_0)$ expresses that there is no component with more than one active port: $\forall j :: type_0.\ \bigwedge_{p,q \in \mathbb{P}_0,\ q \neq p} \neg p(j) \vee \neg q(j)$

As expected, the template $\eta^{FOIL}_{interaction}(\mathbb{P}_0)$ restricts FOIL structures to BIP interactions:

▶ **Proposition 5.1.** *Let $\mathbb{P}_0$ be a set of ports, and η be the instantiation of $\eta^{FOIL}_{interaction}$ with $\mathbb{P}_0$. A FOIL structure ξ satisfies η if and only if ξ induces an interaction.*

To express that a component has at least one active port, we introduce template $active(j) \equiv \bigvee_{p \in \mathbb{P}_0} p(j)$. To simplify notation, parameterization of $active(j)$ by $\mathbb{P}_0$ is omitted.

5.2 Pairwise rendezvous in a clique

In a BIP model, components are said to communicate by binary rendezvous, if all the allowed interactions consist of exactly two ports. The communication is said to be by *pairwise rendezvous*, if there is a binary rendezvous between every two components. Pairwise rendezvous has been widely used as a basic primitive in the parameterized model checking literature, e.g., in [18, 3].

FOIL-templates. We construct a template using two formulae $\eta^{FOIL}_{\leq 2}(\mathbb{P}_0)$ and $\eta^{FOIL}_{\geq 2}(\mathbb{P}_0)$:

- The formula $\eta^{FOIL}_{\leq 2}(\mathbb{P}_0)$ expresses that every interaction has at most two ports: $\forall i, j, \ell :: type_0.\ active(i) \wedge active(j) \wedge active(\ell) \rightarrow i = j \vee j = \ell \vee i = \ell$.
- The formula $\eta^{FOIL}_{\geq 2}(\mathbb{P}_0)$ expresses that every interaction has at least two ports: $\exists i, j :: type_0 : i \neq j.\ active(i) \wedge active(j)$.

We show that the combination of $\eta^{FOIL}_{interaction}$, $\eta^{FOIL}_{\geq 2}$, and $\eta^{FOIL}_{\leq 2}$ defines pairwise rendezvous communication in cliques of all sizes:

▶ **Theorem 5.2.** *Given a one-type parameterized* BIP *model $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$, if $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2})$ is valid, then for every instance $\mathbb{B}^{N,\Gamma}$, the following holds:*

1. *every interaction is of size 2, that is, $|\gamma| = 2$ for $\gamma \in \Gamma$, and*
2. *for every pair of indices i and j such that $0 \leq i, j < N$ and $i \neq j$ and every pair of ports $p, q \in \mathbb{P}_0$, there is a FOIL structure ξ such that $\xi \models_{\text{FOIL}} \psi \wedge p(i) \wedge q(j)$.*

Proof. Fix an instance $\mathbb{B}^{N,\Gamma}$ of $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$.

To show Point 1, fix an interaction γ of $\mathbb{B}^{N,\Gamma}$. By Definition 3.3, there is a FOIL structure ξ such that $\xi \models_{\text{FOIL}} \psi$ and $\gamma = \gamma_\xi$. As ξ induces an interaction, by Proposition 5.1, we immediately have that γ_ξ satisfies the instantiation of $\eta^{FOIL}_{interaction}$. Hence, since $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2})$ is valid we conclude that ξ also satisfies $\eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2}$. This immediately gives us the required equality $|\gamma_\xi| = 2$.

To show Point 2, fix a pair of indices i and j such that $0 \le i, j < N$ and $i \ne j$ and a pair of ports $p, q \in \mathbb{P}_0$. The set $\gamma = \{(p, i), (q, j)\}$ is an interaction. Obviously, one can construct a FOIL structure ξ that induces γ. Since $i \ne j$ and $|\gamma_\xi| = 2$, it holds that $\xi \models_{\text{FOIL}} \eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\ge 2} \wedge \eta^{FOIL}_{\le 2}$. Thus, since $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\ge 2} \wedge \eta^{FOIL}_{\le 2})$ is valid, it follows that $\xi \models_{\text{FOIL}} \psi$. From this and that ξ induces the interaction γ, we conclude that $\xi \models_{\text{FOIL}} \psi \wedge p(i) \wedge q(j)$. ◀

In Theorem 5.2, the right-hand side of the equivalence does not restrict which pairs of ports may interact, e.g., it does not require the ports to be the same. Thus, if ψ is more restrictive than the right-hand side of the equivalence, validity will not hold. Obviously, one can further restrict the equivalence to reflect additional constraints on the allowed pairs of ports. Moreover, one may restrict which ports are required by the template to communicate via pairwise rendezvous for compositionality, e.g. to allow other ports to participate in other communication primitives and in internal transitions. (One may augment or restrict the templates of all the architectures below similarly.)

Applications. Theorem 5.2 gives us a criterion for identifying parameterized BIP models, where all processes may interact with each other using rendezvous communication. To verify such parameterized BIP models, we can immediately invoke the seminal result by German & Sistla [18, Sec. 4]. Their result applies to specifications written in indexed linear temporal logic without the operator $\mathbf{X}$.

More formally, we say that an $ICTL^\star$ path formula $\chi(i)$ is a 1-$LTL\backslash X$ formula, if χ has only one index variable i and χ does not contain quantifiers $\exists$, $\forall$, $\mathbf{A}$, $\mathbf{E}$, nor temporal operator $\mathbf{X}$. Given a parameterized BIP model $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$ and a 1-$LTL\backslash X$ formula χ, one can check in polynomial time, whether every instance $\mathbb{B}^{N,\,\Gamma}$ satisfies the formula $\mathbf{E}\ \exists i :: type_0 :\ true.\ \chi(i)$.

5.3 Broadcast in a clique

In BIP, components communicate via broadcast, if there is a "trigger" component whose sending port is active, and the other components either have their receiving port active, or have no active ports. In this section, we denote the sending port with *send* and the receiving port with *receive*. Our results can be easily extended to treat multiple sending and receiving ports. In a broadcast step, all the components with the active ports make their transitions simultaneously. Broadcasts were extensively studied in the parameterized model checking literature [17, 23].

One way to enforce all the processes to receive a broadcast, if they are ready to do so, is to use priorities in BIP: an interaction has priority over any of its subsets. In this paper, we consider BIP without priorities. In this case, one can express broadcast by imposing the following restriction on the structure of the component type $\mathbb{B}$: *every location has a transition labeled with the port receive*. This restriction enforces all interactions to involve all the components, though some of the components may not change their location by firing a self-loop transition. This requirement can be statically checked on the transition system of $\mathbb{B}$, and if the component type does not fulfill the requirement, it is easy to modify the component type's transition system by adding required self-loops.

FOIL-templates. First, we define the formula $\eta^{FOIL}_{bcast}(\mathbb{P}_0)$, which guarantees that every interaction includes one sending port by one component and the receiving ports of the other components:

$$\exists i :: type_0.\ send(i) \wedge \forall j :: type_0 : j \ne i.\ receive(j)$$

We show that the combination of $\eta^{FOIL}_{interaction}$ and η^{FOIL}_{bcast} defines broadcast in cliques of all sizes:

▶ **Theorem 5.3.** *Given a one-type parameterized* BIP *model* $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$, *if* $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{bcast})$ *is valid, then for every instance* $\mathbb{B}^{N,\Gamma}$*, the following holds:*

1. *every interaction consists of one send port and* $N-1$ *receive ports.*
2. *for every index* c*, such that* $0 \leq c < N$*, there is a FOIL structure* ξ *satisfying the following:*
   $\xi \models_{\text{FOIL}} \psi \wedge send(c) \wedge \forall j :: type_0 : j \neq c.\ receive(j)$.

Proof. The proof follows the same principle as the proof of Theorem 5.2. ◀

Applications. Theorem 5.3 gives a criterion for identifying parameterized BIP models in which all components may send and receive broadcast. Its implications are two-fold. First, it is well-known that parameterized model checking of safety properties is decidable [1] (cf. the discussion in [17]), and there are tools for well-structured transition systems applicable to model checking of parameterized BIP. Second, parameterized model checking of liveness properties is undecidable [17]. From the user perspective, this indicates the need to construct abstractions, or to use semi-decision procedures.

Identifying sending and receiving ports. Now we illustrate how to automatically detect the sending and receiving ports in a parameterized BIP model. We say that a port $p \in \mathbb{P}_0$ in the component type may be a sending port, if in every interaction exactly one component uses this port. Similarly, we say that a port $q \in \mathbb{P}_0$ in the component type may be a receiving port, if in every interaction all but one component use this port. Intuitively, we have to enumerate all port types and check whether they are acting as sending ports or receiving ports. Formally, to find whether p is a potential sending port and q is a potential receiving port, we check whether the following is valid:

$$\psi \;\wedge\; \eta^{FOIL}_{interaction} \;\wedge\; \exists i :: type_0.\ \big(p(i) \vee q(i)\big) \;\rightarrow\; \big(\exists i :: type_0.\ p(i) \wedge \forall j :: type_0 : j \neq i.\ q(j)\big)$$

5.4 Token rings

Token ring is a classical architecture: (i) all processes are arranged in a ring, (ii) the ring size is parameterized but fixed in each run, and (iii) one component owns the token and can pass the token to its neighbor(s). It is easy to express token-passing with rendezvous, so we re-use the templates from Section 5.2. We assume that there is a pair of ports: the port *send* giving away the token and the port *receive* accepting the token. We do not allow the token to change its type, as the parameterized model checking problem is undecidable in this case [26, 16]. Nevertheless, it is easy to extend our results to multiple token types. Here the token is passed in one direction, that is, every component may only receive the token from one neighbor and may only send the token to its other neighbor.

TL-templates. Following the standard assumption [16], we require that every process sends and receives the token infinitely often. We encode this requirement as a local constraint in a form of an LTL formula that is checked against the component type (and not against a BIP instance):

$$\mathbf{G}\Big(receive \rightarrow \mathbf{X}\ (\neg receive\ \mathbf{U}\ \ send)\Big) \wedge \mathbf{G}\Big(send \rightarrow \mathbf{X}\ (\neg send\ \mathbf{U}\ \ receive)\Big)$$

The left conjunct forces a component that has the token to eventually send it. The right conjunct prevents a component from sending the token twice before receiving it back.

FOIL-templates. We extend the pairwise rendezvous templates with a formula $\eta^{FOIL}_{unirlng}(\mathbb{P}_0)$ that restricts the interactions to be performed only among the neighbors in one direction:

$\exists i, j :: type_0.\ (j = (i+1) \bmod n_0).\ send(i) \wedge receive(j)$

The modulo notation "$j = (i+1) \bmod n_0$" can be seen as syntactic sugar, as it expands into $(i = n_0 - 1 \to j = 0) \wedge (i < n_0 - 1 \to j = i+1)$.

▶ **Theorem 5.4.** *Given a one-type parameterized* BIP *model* $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$*, if* $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2} \wedge \eta^{FOIL}_{unirlng})$ *is valid, then every instance* $\mathbb{B}^{N,\Gamma}$ *satisfies:*

1. *every interaction* $\gamma \in \Gamma$ *is of the form* $\{send(c), receive(d)\}$ *for some indices c and d such that* $0 \leq c, d < N$ *and* $d = (c+1) \bmod N$*, and*
2. *for every index c such that* $0 \leq c < N$ *and the index* $d = (c+1) \bmod N$*, there is a FOIL structure* ξ *such that* $\xi \models_{\text{FOIL}} \psi \wedge send(c) \wedge receive(d)$*.*

Proof. The proof follows the same principle as the proof of Theorem 5.2. ◀

Distributing the token. The token ring architecture assumes that initially only one component has the token. Emerson & Namjoshi [16] assumed that the token was distributed using a "daemon", but this primitive is obviously outside of the token ring architecture. Our framework encompasses token distribution. To this end, we restrict the transition system of the component as follows:

- We assume that the location set $\mathbb{Q}_0$ of the component type $\mathbb{B}_0$ is partitioned into two sets: $\mathbb{Q}_0^{tok}$ is the set of locations possessing the token, and $\mathbb{Q}_0^{ntok}$ is the set of locations without the token. The initial location does not possess the token: $\ell^0 \in \mathbb{Q}_0^{ntok}$.
- We assume that there are two auxiliary ports called *master* and *slave* that are only used in a transition from the initial location ℓ^0. There are only two transitions involving ℓ^0: the transition from ℓ^0 to a location in $\mathbb{Q}_0^{tok}$ that broadcasts via the port *master*, and the transition from ℓ^0 to a location in $\mathbb{Q}_0^{ntok}$ that receives the broadcast via the port *slave*. The broadcast interaction can be checked with the constraints similar to those in Section 5.3.

Applications. Theorem 5.4 gives us a criterion for identifying parameterized BIP models that express a unidirectional token ring. This criterion has a great impact: one can apply non-parameterized BIP tools to verify parameterized BIP designs expressing token rings. As Emerson & Namjoshi showed in their celebrated paper [16], to verify parameterized token rings, it is sufficient to run model checking on rings of small sizes. The bound on the ring size—called a *cut-off*—depends on the specification and typically requires two or three components.

5.5 Pairwise rendezvous in a star

In a star architecture, one component acts as the center, and the other components communicate only with the center. The components communicate via rendezvous (considered in Section 5.2). This architecture is used in client-server applications. Parameterized model checking for the star architecture was investigated by German & Sistla [18]. We assume that a parameterized BIP model contains two component types: $\mathbb{B}_0$ with only one instance, and $\mathbb{B}_1$ that may have many instances.

Table 1 Experimental results of identifying architecture models. The column “Outcome” indicates, whether the benchmark was recognized to have the given architecture (positive), or not (negative). The experiments were performed on a 64-bit Linux machine with 2.8GHz $\times$ 4 CPU and 7.8GiB memory.

Benchmark	Architecture model	Outcome	Time (sec.)	Memory (MB)
Milner’s scheduler	uni-directional token ring	**positive**	0.068	$\leq$ 10
Milner’s scheduler	broadcast in clique	negative	0.016	$\leq$ 10
Semaphore	pairwise rendezvous in star	**positive**	0.096	$\leq$ 10
Semaphore	pairwise rendezvous in clique	negative	0.084	$\leq$ 10
Barrier	broadcast in clique	**positive**	0.028	$\leq$ 10
Barrier	pairwise rendezvous in star	negative	0.008	$\leq$ 10

FOIL-templates. The requirements of rendezvous communication are defined in Section 5.2. We add the restriction η^{FOIL}_{center} that the center is involved in every interaction: $\exists i :: type_0.\ active_0(i)$. By restricting ϵ to have only one instance of type $\mathbb{B}_0$, we arrive at Theorem 5.5, which to a large extent is a consequence of Theorem 5.2.

▶ **Theorem 5.5.** *Given a two-component parameterized* BIP *model* $\langle\langle\mathbb{B}_0, \mathbb{B}_1\rangle, \langle n_0, n_1\rangle, \psi, \epsilon\rangle$, *if* $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2} \wedge \eta^{FOIL}_{center})$ *and* $\epsilon \leftrightarrow (n_0 = 1)$ *are both valid, then every instance* $\langle\mathbb{B}_0, \mathbb{B}_1\rangle^{\langle N_0, N_1\rangle, \Gamma}$ *admits only the rendezvous interactions with the center, i.e., the only component of type* $\mathbb{B}_0$.

Applications. Theorem 5.5 gives us a criterion for identifying parameterized BIP models, where the user processes communicate with the coordinator via rendezvous. To verify such parameterized BIP models, we can immediately invoke several results by German & Sistla [18, Sec. 3]. First, one can analyze such parameterized BIP models for deadlocks, which is of extreme importance to the practical applications of BIP. Second, the results [18] reduce parameterized model checking to reachability in Petri nets, which allows one to use the existing tools for Petri nets.

6 Prototype implementation and experiments

We have implemented a prototype of the framework introduced in Section 5. This prototype uses the following architecture templates: (a) pairwise rendezvous and broadcast in cliques, (b) token rings, (c) and pairwise rendezvous in stars. As described in Section 5 (see *our framework*), given a parameterized BIP model, the tool constructs a set of FOIL formulae and a set of temporal formulae. The parameterized BIP model follows a predefined architecture, if the FOIL formulae are valid and the component types satisfy the temporal formulae. Our implementation uses nuXmv [11] to check temporal formulae and Z3 [14] to check validity of first-order formulae. FOIL formulae are translated to first-order formulae by guarding the range of quantification explicitly, e.g. $\exists i :: type_0.\ \theta$ is substituted with $\exists i.\ 0 \leq i < n_0 \wedge \theta$. To deal with quantifiers, we run a customized solver with tactic ’qe’ (i.e. quantifier elimination). The implementation and benchmarks are available at `http://risd.epfl.ch/parambip`.

Table 1 summarizes our experiments with three benchmarks. We conducted each experiment using two kinds of templates: the expected architecture of the benchmark, and an architecture different from the expected one. In all cases, the architectures were identified as expected. Our preliminary results demonstrate both correctness and efficiency of our approach.

7 Related work and conclusions

We have shown that our framework encompasses several prominent parameterized model checking techniques. To our understanding, the other seminal results can be integrated into our framework: the cut-off results for disjunctive and conjunctive guards [15], network decomposition techniques [13, 3], and techniques based on well-structured transition systems [1] and monotonic abstraction [2].

First-order interaction logic extends propositional interaction logic [6, 7], which was also extended by Dy-BIP [10] and configuration logic [21]. Dy-BIP extends propositional interaction logic with quantification to define interaction topology independent of the number of component instances. It uses dedicated *history variables* to break the symmetry and specify that, throughout the system execution, successive interactions happen among the same components. Dy-BIP does not have a mechanism, such as indexing, to statically distinguish instances of the same component type. Hence, many architectures, e.g., token rings, cannot be expressed. Configuration logic uses higher-order formulae to represent sets of topologies. It does not use indexing either, thereby requiring the second-order extension to express simple architectures such as token rings and linear architectures. Finally, no decidability results or decision procedures have been proposed for the configuration logic yet.

In the future, we will study second-order extensions of FOIL to express more complex architectures such as server-client whose coordinator is chosen non-deterministically. In the long term, we plan to implement a tool that integrates multiple parameterized model checking techniques and uses our framework to guide the verification of parameterized BIP designs. FOIL can also be seen as a specification language for BIP interactions and used for their synthesis similarly to [7]. Finally, it is worth investigating, whether FOIL can be extended to include priorities as in [8].

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay. General decidability theorems for infinite-state systems. In *LICS*, 1996.
2. P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Monotonic abstraction: on efficient verification of parameterized systems. *Int. J. Found. Comput. Sci.*, 2009.
3. B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR*. Springer, 2014.
4. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *Software, IEEE*, 2011.
5. S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, W. Saab, and Q. Wang. Formal verification of infinite-state BIP models. In *ATVA*, 2015.
6. S. Bliudze and J. Sifakis. The algebra of connectors —structuring interaction in BIP. In *EMSOFT*, 2007.
7. S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. *FMSD*, 2010.
8. S. Bliudze and J. Sifakis. Synthesizing glue operators from glue constraints for the construction of component-based systems. In *Software Composition*, 2011.
9. R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. Decidability of parameterized verification. *Synthesis Lectures on Distributed Computing Theory*, 2015.
10. M. Bozga, M. Jaber, N. Maris, and J. Sifakis. Modeling dynamic architectures using Dy-BIP. In *Software Composition*. Springer, 2012.
11. 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 *CAV*, 2014.

12 E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

13 E. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *CONCUR*, 2004.

14 L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

15 E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *LICS*, 2003.

16 E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *POPL*, 1995.

17 J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. *LICS*, 1999.

18 S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 1992.

19 O. Grumberg and H. Veith. *25 years of model checking: history, achievements, perspectives.* Springer, 2008.

20 J. Y Halpern. Presburger arithmetic with unary predicates is π_1^1 complete. *J. of Symb. Logic*, 1991.

21 A. Mavridou, E. Baranov, S. Bliudze, and J. Sifakis. Configuration logics: Modelling architecture styles. In *FACS*, 2015.

22 Q.Wang and S. Bliudze. Verification of component-based systems via predicate abstraction and simultaneous set reduction. In *TGC*, 2015.

23 S. Schmitz and P. Schnoebelen. The power of well-structured systems. In *CONCUR*, 2013.

24 J. Sifakis. Rigorous system design. *Foundations and Trends in Electr. Design Automation*, 2013.

25 J. Sifakis. System design automation: Challenges and limitations. *Proc. of the IEEE*, 2015.

26 I. Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 1988.

27 W. Thomas. *Languages, automata, and logic.* Springer, 1997.

Part V

Supplementary Documents

Chapter 9

Curriculum Vitae

Igor KONNOV

INRIA Nancy & LORIA
Equipe VeriDis, Bâtiment B
615, rue du Jardin Botanique
F-54602 Villers-lès-Nancy, France
Homepage: http://forsyte.at/konnov
Phone: +33 (0)3-83-59-30-75

Born in 1981. Married, with one child.
Languages: English (fluent), German (ÖSD Zertifikat B2), Russian (mother tongue), French (beginner)
Research interests: software verification, model checking, parameterized model checking, verification of distributed algorithms, temporal logic of actions (TLA^+)

Appointments

INRIA Nancy — Grand Est, Nancy, France:

Mar 2018 – present. Researcher (permanent, chargé de recherche)

Vienna University of Technology (TU Wien), Faculty of Informatics, Austria:

Jan 2016 – Feb 2018. Postdoctoral researcher, principal investigator in the WWTF project APALACHE
Dec 2011 – Dec 2015. Postdoctoral assistant professor (Universitätsassistent, limited contract)
Jul 2011 – Dec 2011. Postdoctoral researcher (Projektassistent)

Moscow State University (MSU), Faculty of Computational Mathematics and Cybernetics, Russia:

Jan 2010 – Jun 2011. Junior research fellow (m.n.s.)
Dec 2006 – Jan 2010. Pre- and postdoctoral research and teaching assistant

Sytech LLC, Russia: **2006–2010,** Part-time systems architect, **2004–2006,** Software developer

IFirst LLC, Russia: **01.09.2002–15.09.2003,** Part-time programmer

Higher education

Oct 2003– Nov 2008. Moscow State University, Russia:
Ph.D. in Computer Science (awarded in **Feb 2009**)

Sep 1998– Jul 2003. Moscow State University, Russia:
Specialist (approx. MSc) in Applied Math. & Informatics

With distinction, 97% are the best score: avg. score 1.06 (German scale) = avg. score 4.87 (Russian scale)

Project acquisition and participation

2016–2019. WWTF: Vienna Science and Technology Fund. Project ICT15–103 APALACHE **539k€**
Abstraction-based Parameterized TLA Checker TU Wien
Role: principal investigator, **with**: J.Widder (co-PI), H. Veith (core team)
Acceptance rate: 10 out of 137 proposals (approx. 7%)

2015–2018. FWF: Austrian National Research Network S11403-N23 SHiNE
Systematic Methods in Systems Engineering TU Wien
Role: researcher, **Coordinator**: R. Bloem (**3.7 Mio.€**), **PI**: H. Veith **625k€**

2011–2014. WWTF: Vienna Science and Technology Fund. Project PROSEED **598k €**
Proof Seeding for Software Verification TU Wien
Role: researcher, **PI**: H. Veith

2010–2014. FWF: Austrian National Research Network S11403-N23 RiSE
Rigorous Systems Engineering TU Wien
Role: researcher, **Coordinator**: R. Bloem (**3.7 Mio. €**), **PI**: H. Veith **582.8k €**

2010–2012. Russian Federal Special-Purpose Programme, Project 14.740.11.0399 approx. **200k €**
Developing a Prototype for Computer Simulation of Real-Time Distributed Systems MSU
Role: responsible for coordination, research agenda, and report writing, **PI**: R.L. Smeliansky

2009–2011. RFBR: Russian Fund for Basic Research, Project Nr. 09–01–00277-a approx. **32k €**
Structural and Semantic Analisys Using Formal Models of Sequential and Parallel Processes MSU
Role: researcher, **PI**: R.I. Podlovchenko

2006–2009. INTAS: EU research cooperation with the New Independent States, Project Nr. 05–1000008–8144
Practical Formal Verification Using Automated Reasoning and Model Checking MSU
Role: researcher, **Coordinator**: T. Jebelian, **PI**: V.E. Plisko

2006–2008. RFBR: Russian Fund for Basic Research, Project Nr. 06–01–00106-a approx. **52k €**
Formal Models of Sequential and Parallel Processes and the Analysis of Their Semantic Properties MSU
Role: researcher, **PI**: R.I. Podlovchenko

R&D projects with industry and state companies

2009–2010. *Obfuscation techniques on intermediate code representation* Computer Systems Lab/MSU
Role: team lead of 1 master student and 1 PhD student, **PI:** R.L. Smeliansky

2007–2008. *Obfuscation techniques for C++* Computer Systems Lab/MSU
Role: team lead of 1 master student and 1 PhD student, **PI:** R.L. Smeliansky

2008. *Teachable static analysis workbench* The Open Web Application Security Project (OWASP)
Role: developer, **PI:** D.D. Kozlov

2007–2008. *Static analysis of python web applications for vulnerabilities* Computer Systems Lab/MSU
Role: developer, **PI:** R.L. Smeliansky

Selected invited talks & lectures

Dagstuhl Seminar 18211: "Formal Methods and Fault-Tolerant Distributed Computing: Forging an Alliance"
Dagstuhl/Germany, invited tutorial *What my computer can find about your distributed algorithm* **May 2018**

Bertrand Meyer's Vericlub seminar, U. Toulouse, Toulouse/France **Nov 2016**
invited seminar talk *Model checking of threshold-guarded distributed algorithms: beyond reachability*

Rigorous System Design Laboratory, EPFL, Lausanne/Switzerland **Sep 2016**
invited seminar talk *Model checking of fault-tolerant distributed algorithms: safety and liveness*

Workshop on Program Semantics, Specification & Verification at CSR'16, St. Petersburg/Russia
invited talk *Model checking of threshold-based fault-tolerant distributed algorithms* **Jun 2016**

Spring School Logic & Verification, Vienna/Austria **Apr 2016**
lectures on *Complete parameterized & bounded model checking of threshold-based fault-tolerant distributed algorithms*

Amazon, Herndon, VA/USA **Jun 2015**
invited talk *Model checking of threshold-based fault-tolerant distributed algorithms*

Dagstuhl Seminar: "Distributed Cloud Computing", Dagstuhl/Germany **Feb 2015**
talk *Model checking of threshold-based fault-tolerant distributed algorithms*

Tools & Methods of Program Analysis'14, Kostroma/Russia **Nov 2014**
invited talk *Parameterized model checking of fault-tolerant distributed algorithms by abstraction*

Summer School'14: "Verification Technology, Systems & Applications", Luxembourg **Oct 2014**
lectures on *Model checking of fault-tolerant distributed algorithms* (together with Helmut Veith)

Dagstuhl Seminar: "Formal Verification of Distributed Algorithms", Dagstuhl/Germany **Apr 2013**
invited talk *Counter attack on Byzantine generals*

Concurrency Seminar, Computing Laboratory, Oxford/UK **Feb 2011**
invited talk *An invariant-based approach to the verification of asynchronous parameterized networks*

Teaching experience

Vienna University of Technology (TU Wien)

2013–2017. Computer Aided Verification Master students, compulsory, lectures & practicals, 3 ECTS
In 2017, held the lecture course. Until 2017, read parts of the lecture course, teaching assistance.

2013–2017. Program & Systems Verification Bachelor students, compulsory, lectures & practicals, 6 ECTS
Teaching assistance

2011–2015. Formal Methods of Informatics Master students, compulsory, lectures & practicals, 6 ECTS
Teaching assistance

Moscow State University (MSU)

2008–2010. Software model checking (Dr. Savenkov) 8th semester, compulsory, lectures & seminars, 32 hrs.
Designed the course together with K. Savenkov, read parts of the lecture course, teaching assistance

2004. Seminars on The C Programming Language and UNIX 3rd semester, compulsory, 32 hrs.
Instructed at all seminars (approx. 20 students)

2005. Seminars on Syntax Analysis and C++ 4th semester, compulsory, 32 hrs.
Instructed at all seminars (approx. 20 students)

2004. Operating Systems (Prof. Terekhov) 3rd semester, compulsory, lectures, 54 hrs.
Teaching assistance

2003–2011. Computer Networks (Prof. Smeliansky) 6th semester, compulsory, lectures, 64 hrs.
Teaching assistance

2003–2004. The Java Programming Language optional, lectures, 32 hrs.
Read parts of the lecture course, teaching assistance

2003–2004. MSU *math entrance exams* compulsory
Corrected written math exams, participated in the oral math exams

Kazakhstan branch of Moscow State Univ., Astana/Kazakhstan

2011. Software model checking 8th semester, compulsory, lectures & seminars, 32 hrs.
held the lecture course and the seminars

Tashkent University, Tashkent/Uzbekistan

2011–2013. Participated in EU project CANDI: Teaching Competency & Infrastructure for e-Learning and Re-training

Advising

PhD students (TU Wien): Associated Faculty of Doctoral College LogiCS [`logic-cs.at`]

2016–present. Thanh Hai Tran (advising) with Priv.-Doz. Dr. Josef Widder

2016–present. Jure Kukovec (advising) with Priv.-Doz. Dr. Josef Widder

2015–present. Marijana Lazić (co-advising) with Prof. Roderick Bloem, Priv.-Doz. Dr. Josef Widder

2011–2014. Annu Gmeiner (informal co-advising)
Parameterized model checking of fault-tolerant distributed algorithms advisor: Prof. Helmut Veith

Master students:

2016. Jure Kukovec (Univ. Ljubljana)
Extensions of Threshold Automata for Reachability in Parameterized Systems co-advised with Prof. Andrej Bauer

2015–2016. Thanh Hai Tran (TU Wien)
User-guided Predicate Abstraction of TLA+ Specifications co-advised with Prof. Helmut Veith

2009–2011. Alexander Mischenko (MSU)
Static Type Analysis of Python Programs on Bytecode Level

2007–2009. Denis Sigaev (MSU)
Detection of Programs Protected from Reverse Engineering co-advised with A. Kachalin

2008. Alexey Schevchenko (MSU)
Application of Regular Model Checking to Infinite State Systems

2007. Peter Bulychev (MSU)
Game-Theoretic Methods of Protocol Verification co-advised with Prof. Vladimir Zakharov

Bachelor students:

2013. Sebastian Neumaier (TU Wien) *A Simple Simulation Language for Distributed Algorithms*

2011. Andrey Babak and Anton Artyomov (MSU) *Static Analysis of Python Programs*

Community service

Program Committees:

ACM Symposium on Principles of Distributed Computing (PODC'18) London/UK

Formal Methods in Computer-Aided Design (FMCAD) 2017 & 2018 — Vienna/Austria & Austin/TX, US
Computer Aided Verification'16 (External Reviewer Committee) — Toronto/Canada
Symbolic and Numeric Algorithms for Scientific Computing 2013, 2016, and 2017 — Timisoara/Romania
Stabilization, Safety, and Security of Distributed Systems'15 — Edmonton/Canada
Intl. Conf. on Verification & Evaluation of Computer & Comm. Systems (VECoS'18) — Grenoble/France
Intl. Symposium on Formal Approaches to Parallel & Distributed Systems (4PAD) 2018 — Orléans/France
Workshop on Methods and Tools for Rigorous System Design (MeTRiD'18) — Thessaloniki/Greece
Tools & Methods of Program Analysis 2015 & 2017 — St. Petersburg & Moscow/Russia
Workshop on Program Semantics, Specification, and Verification 2017 & 2018 — Moscow & Yaroslavl/Russia
Parallel, Distributed, and Network-based Processing'17 (Formal approaches track) — St. Petersburg/Russia

Artifact Evaluation Chair: Computer-Aided Verification (CAV'18) [`cavconference.org/2018/`]

Journal and book chapter reviews: FMSD (2018), LMCS (2017), ACM ToCL (2017), MAIS (2017), TIME (2015),
Handbook of Model Checking (eds. E. Clarke, T. Henzinger, H. Veith)

Guest editor: Special issue on Computer Aided Verification'13 in Formal Methods in System Design (Springer) (with Helmut Veith and Natasha Sharygina)

Editorial board: Proceedings of the Institute for System Programming of the Russian Academy of Sciences since 2016 [`www.ispras.ru/en/proceedings`]

External reviewer: FSTTCS'17, QEST'17, TACAS'17, STACS'17, VMCAI'17, MARS'17, ICFEM'16, CONCUR'16, IJCAR'16, LICS'16, EuroPar'16, AAMAS'16, CAV'15, FMCAD'15, TACAS'15, FoSSaCS'15, CAV'14, SAS'14, GandALF'14, ESOP'14, HVC'14, CAV'13, LATA'13, SSS'13, CAV'12, NFM'12, SPIN'12, VMCAI'12, FMICS'11, CSL'11

Workshop chair of CAV'13. Conference on Computer Aided Verification [`cav2013.forsyte.at`]

Co-organizer of FRIDA'14–18. Workshop on Formal Reasoning in Distributed Algorithms:

FRIDA'18, July 13, 2018. Co-located with CAV'18 [`forsyte.at/events/frida2018`]
FRIDA'17, October 16, 2017. Co-located with DISC'17 [`forsyte.at/events/frida2017`]
FRIDA'16, May 17, 2016. Co-located with NETYS'16 [`forsyte.at/events/frida2016`]
FRIDA'15, June 5, 2015. Co-located with FORTE'15 [`discotec2015.inria.fr/workshops/frida-2015/`]
FRIDA'14, July 23–24, 2014. Co-located with CAV'14 [`vsl2014.at/frida/`]

Student Award Committee. VCLA International Student Awards 2014–2015

Tools

2012–present. ByMC: model checker of parameterized fault-tolerant distributed algorithms [`forsyte.at/software/bymc`]

2004–2009. CheAPS: model checker of parameterized asynchronous distributed systems [`lvk.cs.msu.su/~konnov/cheaps`]

Chapter 10

List of Publications

Publications by Igor Konnov

Book

[1] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification.* Vol. 6. 1. Morgan & Claypool, 2015, pp. 1–170. DOI: 10.2200/S00658ED1V01Y201508DCT013.

Book chapter

[2] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. "Tutorial on Parameterized Model Checking of Fault-Tolerant Distributed Algorithms". In: *Formal Methods for Executable Software Models.* LNCS. Springer, 2014, pp. 122–171. DOI: 10.1007/978-3-319-07317-0_4.

Invited paper

[3] I. Konnov, H. Veith, and J. Widder. "What You Always Wanted to Know About Model Checking of Fault-Tolerant Distributed Algorithms". In: *Perspectives of System Informatics: PSI 2015, in Memory of Helmut Veith, Revised Selected Papers.* Springer, 2016, pp. 6–21. DOI: 10.1007/978-3-319-41579-6_2.

Journal articles

[4] I. V. Konnov, H. Veith, and J. Widder. "On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability". In: *Information and Computation* 252 (2017). **(Extended version of the conference paper** I. Konnov, H. Veith, J. Widder. "On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability". In Concurrency Theory – 25th International Conference, CONCUR, 2014, pp. 125–140**)**, pp. 95–109. DOI: 10.1016/j.ic.2016.03.006.

[5] I. Konnov, M. Lazic, H. Veith, and J. Widder. "Para^2: Parameterized Path Reduction, Acceleration, and SMT for Reachability in Threshold-Guarded Distributed Algorithms". In: *Formal Methods in System Design* (2017). **(Extended version of the conference paper** I. Konnov, H. Veith, J. Widder. "SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms". In Computer-Aided Verification, vol. 9206, LNCS, 2015, pp. 85–102.**)** DOI: 10.1007/s10703-017-0297-4. URL: https://link.springer.com/article/10.1007/s10703-017-0297-4.

[6] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. "Decidability in Parameterized Verification". In: *ACM SIGACT News* 47.2 (2016), pp. 53–64. DOI: 10.1145/2951860.2951873.

[7] D. Y. Volkanov, V. A. Zakharov, D. A. Zorin, V. V. Podymov, and I. V. Konnov. "A combined toolset for the verification of real-time distributed systems". In: *Programming and Computer Software* 41.6 (2015), pp. 325–335. DOI: 10.1134/S0361768815060080.

[8] I. Konnov, V. Podymov, D. Volkanov, V. Zakharov, and D. Zorin. "How to Make a Simple Tool for Verification of Real-Time Systems". In: *Automatic Control and Computer Sciences* 48.7 (2014), pp. 534–542. DOI: 10.3103/S0146411614070232.

[9] I. V. Konnov. "On application of weaker simulations to parameterized model checking by network invariants technique". In: *Automatic Control and Computer Sciences* 44.7 (2010), pp. 378–386. DOI: 10.3103/S0146411610070035.

[10] I. V. Konnov and V. A. Zakharov. "An invariant-based approach to the verification of asynchronous parameterized networks". In: *Journal of Symbolic Computation* 45.11 (2010), pp. 1144–1162. DOI: 10.1016/j.jsc.2008.11.006.

[11] I. V. Konnov and V. A. Zakharov. "Using Adaptive Symmetry Reduction for LTL Model Checking". In Russian. In: *Modelling and Analysis of Information Systems* 17.4 (2010), pp. 78–87. URL: http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=mais&paperid=38&option_lang=eng.

[12] I. V. Konnov and V. A. Zakharov. "An Approach to the Verification of Symmetric Parameterized Distributed Systems". In: *Programming and Computer Software* 31.5 (2005), pp. 225–236. DOI: 10.1007/s11086-005-0034-4.

Peer-reviewed conference proceedings

[13] I. Konnov and J. Widder. "ByMC: Byzantine Model Checker". In: *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems.* Cham: Springer International Publishing, 2018, pp. 327–342. DOI: 10.1007/978-3-030-03424-5_22. URL: https://hal.inria.fr/hal-01909653.

[14] J. Kukovec, I. Konnov, and J. Widder. "Reachability in Parameterized Systems: All Flavors of Threshold Automata". In: *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. 2018, 19:1–19:17. DOI: 10.4230/LIPIcs.CONCUR.2018.19. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2018.19.

[15] J. Kukovec, T. Tran, and I. Konnov. "Extracting Symbolic Transitions from TLA$^+$ Specifications". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. 2018, pp. 89–104. DOI: 10.1007/978-3-319-91271-4_7. URL: http://forsyte.at/wp-content/uploads/abz2018_full.pdf.

[16] I. V. Konnov, M. Lazic, H. Veith, and J. Widder. "A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017, pp. 719–734. URL: http://dl.acm.org/citation.cfm?id=3009860.

[17] I. V. Konnov, J. Widder, F. Spegni, and L. Spalazzi. "Accuracy of Message Counting Abstraction in Fault-Tolerant Distributed Algorithms". In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. 2017, pp. 347–366. DOI: 10.1007/978-3-319-52234-0_19.

[18] M. Lazic, I. Konnov, J. Widder, and R. Bloem. "Synthesis of Distributed Algorithms with Parameterized Threshold Guards". In: *OPODIS*. Vol. 95. LIPIcs. 2017, 32:1–32:20. URL: https://doi.org/10.4230/LIPIcs.OPODIS.2017.32.

[19] I. Konnov, T. Kotek, Q. Wang, H. Veith, S. Bliudze, and J. Sifakis. "Parameterized Systems in BIP: Design and Model Checking". In: *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. Vol. 59. LIPIcs. 2016, 30:1–30:16. DOI: 10.4230/LIPIcs.CONCUR.2016.30.

[20] I. Konnov, H. Veith, and J. Widder. "SMT and POR beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms". In: *CAV (Part I)*. Vol. 9206. LNCS. 2015, pp. 85–102. DOI: 10.1007/978-3-319-21690-4_6.

[21] I. Konnov, H. Veith, and J. Widder. "On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability". In: *CONCUR 2014*. Vol. 8704. LNCS. 2014, pp. 125–140. DOI: 10.1007/978-3-662-44584-6_10.

[22] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. "Brief announcement: parameterized model checking of fault-tolerant distributed algorithms by abstraction". In: *PODC*. 2013, pp. 119–121. DOI: 10.1145/2484239.2484285.

[23] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. "Parameterized model checking of fault-tolerant distributed algorithms by abstraction". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 2013, pp. 201–209. DOI: 10.1109/FMCAD.2013.6679411.

[24] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. "Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms". In: *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013*. Vol. 7976. LNCS. 2013, pp. 209–226. DOI: 10.1007/978-3-642-39176-7_14.

[25] I. V. Konnov. "Application of CHEAPS System to Parameterized Model Checking of Distributed Systems". In Russian. In: *Proc. 3rd All-Russia Conf. on Methods and Techniques of Information Processing*. Moscow, 2009, pp. 116–122. ISBN: 978-5-89407-373-3.

[26] V. A. Zakharov and I. V. Konnov. "On the Verification of Asynchronous Parameterized Distributed Programs". In Russian. In: *Proc. 2nd All-Russia Conf. on Methods and Techniques of Information Processing*. MAKS Press, Moscow, 2005, pp. 267–372. ISBN: 5-89407-230-1.

[27] I. V. Konnov and V. A. Zakharov. "On the Verification of Parameterized Symmetric Distributed Programs". In Russian. In: *Proc. 1st All-Russia Conf. on Methods and Techniques of Information Processing*. MAKS Press, Moscow, 2003, pp. 395–400. ISBN: 5-89407-163-1.

Peer-reviewed workshop contributions

[28] I. Konnov, J. Kukovec, and T.-H. Tran. *BmcMT: Bounded Model Checking of TLA+ Specifications with SMT*. Contribution to TLA+ Community Meeting, Oxford, UK, July. 2018. URL: http://tla2018.loria.fr/contrib/konnov.pdf.

[29] I. Konnov and S. Merz. *Model Checking of Fault-Tolerant Distributed Algorithms: from Classics towards Contemporary*. Contribution to DSN Workshop on Byzantine Consensus and Resilient Blockchains, Luxembourg City, Luxembourg, June. 2018. URL: https://bcrb18.fim.uni-passau.de/shortpapers/bcrb18-konnov-merz.pdf.

[30] I. Konnov, H. Veith, and J. Widder. *Challenges in Model Checking of Fault-tolerant Designs in TLA+*. Contribution to the 8th International Workshop on Exploiting Concurrency Efficiently and Correctly, San Francisco, CA, USA, July. 2015. URL: http://multicore.doc.ic.ac.uk/events/ec2/KonnovVeithWidder.pdf.

[31] I. Konnov. "CheAPS: a Checker of Asynchronous Parameterized Systems". In: *WING 2010*. Ed. by A. Voronkov, L. Kovacs, and N. Bjorner. Vol. 1. EPiC Series. EasyChair, 2012, pp. 128–129. URL: http://www.easychair.org/publications/?page=355792421.

[32] I. V. Konnov and V. A. Zakharov. "Using Adaptive Symmetry Reduction for LTL Model Checking". In: *Proc. International Workshop on Program Semantics, Specification and Verification (PSSV 2010) affiliated with CSR 2010*. 2010, pp. 5–11. URL: http://csr2010.ksu.ru/PSSV.html.

[33] V. Zakharov and I. Konnov. "An Invariant-based Approach to the Verification of Asynchronous Parameterized Networks". In: *International Workshop on Invariant Generation (WING'07)*. 2007, pp. 41–55. URL: http://www.risc.uni-linz.ac.at/publications/download/risc_3128/proceedings.pdf.

Conference contributions

[34] V. V. Antonenko and I. V. Konnov. "On the Choice of a Simulation Run-Time Infrastructure based on High-Level Architecture". In Russian. In: *17th International Conference on Computational Mechanics and Contemporary Application Software Systems 2011 (VMSPPS'2011), Alushta, Ukraine*. 2011, pp. 36–38. ISBN: 978-5-7035-2269-1.

[35] G. A. Klimov, D. D. Kozlov, and I. V. Konnov. "Static analysis for security of web applications developed in Python". In Russian. In: *Proc. 5th All-Russia Scientific and Technical Conf. Microsoft technologies in theory and practice of programming*. 2008.

[36] I. V. Konnov. "The system for verfication of parameterized models of asynchronous distributed systems (CHEAPS)". In Russian. In: *Proc. 5th All-Russia Scientific and Technical Conf. Microsoft technologies in theory and practice of programming*. 2008.

Workshop contributions

[37] I. Konnov. *Towards symbolic model checking of fault-tolerant designs in TLA+*. Talk at the Helmut Veith Memorial Workshop, Obertauern, Austria, January. 2018. URL: http://hvw2018.cs.uni-salzburg.at/schedule.

[38] I. Konnov. *Verifying Safety and Liveness of Threshold-guarded Fault-Tolerant Distributed Algorithms*. Talk at the Helmut Veith Memorial Workshop, Obergurgl, Austria, February. 2017. URL: http://cbr.uibk.ac.at/events/hvw/schedule.php.

[39] I. Konnov. *SMT and POR beat Counter Abstraction: Parameterized Model Checking of Threshold-based Distributed Algorithms*. Workshop contribution at Alpine Verification Meeting, Attersee, Austria, May. 2015.

[40] A. B. Glonina, I. Konnov, V. V. Podymov, D. Y. Volkanov, V. A. Zakharov, and D. A. Zorin. *An experience on using simulation environment DYANA augmented with UPPAAL for verification of embedded systems defined by UML statecharts*. Contribution to the CAV workshop VES13, St. Petersburg, Russia, July. 2013. URL: http://forsyte.at/wp-content/uploads/ves13-gkpvzz.pdf.

[41] I. Konnov. *Parameterized Model Checking by Network Invariants: the Asynchronous Case*. Contribution to: LICS Workshop AISS, Dubrovnik, Croatia, June 2012. 2012. URL: http://forsyte.at/wp-content/uploads/12konnov-aiss.pdf.

[42] I. Konnov, H. Veith, and J. Widder. *Who is afraid of Model Checking Distributed Algorithms?* Contribution to the 5th International Workshop on Exploiting Concurrency Efficiently and Correctly, Berkeley, CA, USA, July 2012. 7 citations excl. self-citations. 2012. URL: http://forsyte.at/wp-content/uploads/2012/07/ec2-konnov.pdf.

[43] I. V. Konnov and O. Letichevsky. "Model Checking GARP Protocol using Spin and VRS". In: *International Workshop on Automata, Algorithms, and Information Technologies*. 2010. DOI: 10.1007/s10559-010-9244-8.

Technical reports

[44] N. Bertrand, I. Konnov, M. Lazic, and J. Widder. *Verification of Randomized Distributed Algorithms under Round-Rigid Adversaries*. Nov. 2018. URL: https://hal.inria.fr/hal-01925533.

[45] I. Stoilkovska, I. Konnov, J. Widder, and F. Zuleger. *Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking.* working paper or preprint. Nov. 2018. URL: `https://hal.inria.fr/hal-01925653`.

[46] I. Konnov, M. Lazic, H. Veith, and J. Widder. *A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant Distributed Algorithms.* Extended version of the POPL'17 paper including the proofs. 2016. URL: `http://arxiv.org/abs/1608.05327`.

[47] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. *Counter Attack on Byzantine Generals: Parameterized Model Checking of Fault-tolerant Distributed Algorithms.* Oct. 2012. URL: `http://arxiv.org/abs/1210.3846`.

[48] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. *Starting a Dialog between Model Checking and Fault-tolerant Distributed Algorithms.* Oct. 2012. URL: `http://arxiv.org/abs/1210.3839`.

[49] P. Bulychev, I. V. Konnov, and V. A. Zakharov. "Computing (bi)simulation relations preserving CTL^*_{-X} for ordinary and fair Kripke structures". In: *Mathemathical Methods and Algorithms, Institute of Systems Programming of the Russian Academy of Sciences.* Vol. 12. 2006, pp. 59–76. URL: `http://discopal.ispras.ru/pdfs/issue-2006-12/cs-isp-sbornik.pdf`.

[50] I. Konnov and V. Zakharov. "On the verification of asynchronous parameterized networks of communicating processes by model checking". In: *Mathemathical Methods and Algorithms, Institute of Systems Programming of the Russian Academy of Sciences.* Vol. 12. 2006, pp. 37–58. URL: `http://discopal.ispras.ru/pdfs/issue-2006-12/cs-isp-sbornik.pdf`.

Invited Talks

Invited speaker at conferences and workshops

[1] I. Konnov. *Model Checking of Threshold-based Fault-Tolerant Distributed Algorithms.* Invited talk at the 7th Workshop on Program Semantics, Specification & Verification, St. Petersburg, Russia, June. 2016. URL: http://pssv-conf.ru/en/2016/program.

[2] I. Konnov. *Parametrized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction.* Tutorial at the International Conference Tools and Methods of Program Analysis, Kostroma, Russia, November. 2014. URL: http://tmpaconf.org/pasteventsmaterialsen/keynote-speakersen#2014.

Tutorials

[3] I. Konnov. *Model Checking of Fault-tolerant Distributed Algorithms.* Tutorial at the Spring School Logic and Verification, Vienna, April. 2016. URL: http://forsyte.at/events/love2016/.

[4] H. Veith and I. Konnov. *Model Checking of Fault-tolerant Distributed Algorithms.* Tutorial at the Summer School on Verification Technology, Systems & Applications, Luxembourg, Luxembourg, October. 2014. URL: http://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa14/.

Invited Seminar Talks

[5] I. Konnov. *Synthesizing Distributed Algorithms with Parameterized Threshold Guards.* Talk at the Workshop on Verification of Distributed Systems, Essaouira, Morocco, May. 2018. URL: http://netys.net/VDS2018.html.

[6] I. Konnov. *What my computer can find about your distributed algorithm.* Tutorial at the Dagstuhl seminar 18211 "Formal Methods and Fault-Tolerant Distributed Computing: Forging an Alliance", Dagstuhl, Germany, May. 2018. URL: https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=18211.

[7] J. Widder and I. Konnov. *Logical Methods for the Correctness of Distributed Algorithms.* RISE PI talk at "Alpine Verification Meeting", Wagrain, Austria, September. 2018. URL: https://avm2018.iaik.tugraz.at/program/.

[8] I. Konnov. *Model checking of distributed algorithms for LARGE-scale systems.* Interview talk at INRIA (awarded 1 of 4 researcher positions (CR1) at INRIA), Paris, France, May. 2017.

[9] I. Konnov. *Verifying Safety and Liveness of Threshold-guarded Fault-Tolerant Distributed Algorithms.* Talk at LORIA/INRIA seminar, Nancy, France, May. 2017.

[10] I. Konnov. *Model Checking of Fault-tolerant Distributed Algorithms: Safety and Liveness.* Invited talk at the Seminar of Rigorous System Design Laboratory, Lausanne, Switzerland, September. 2016.

[11] I. Konnov. *Model Checking of Threshold-based Fault-tolerant Distributed Algorithms.* Invited talk at the Seminar on Foundations of Mathematics and Theoretical Computer Science, Ljubljana University, Ljubljana, Slovenia, May. 2016.

[12] I. Konnov. *Model Checking of Threshold-Guarded Distributed Algorithms: Beyond Reachability.* Invited talk at the Vericlub Seminar (Bertrand Meyer), Toulouse, France, November. 2016.

[13] I. Konnov. *Model Checking of Threshold-based Fault-tolerant Distributed Algorithms.* Invited talk at Amazon, Herndon, VA, USA, June. 2015.

[14] I. Konnov. *Model checking of threshold-based fault-tolerant distributed algorithms.* Talk at the Dagstuhl Seminar on Distributed Cloud Computing, Dagstuhl, Germany, February. 2015.

[15] I. Konnov. *SMT and POR beat Counter Abstraction.* Invited talk at the RiSE Seminar at Institute of Science and Technology Austria, Klosterneuburg, Austria, April. 2015.

[16] I. Konnov. *On Completeness of Bounded Model Checking for Threshold-based Distributed Algorithms: Reachability.* Talk at the Seminar on Theoretical Problems in Programming, Moscow State University, Moscow, Russia, February. 2014.

[17] I. Konnov. *Counter Attack on Byzantine Generals.* Talk at the Dagstuhl Seminar on Formal Verification of Distributed Algorithms, Dagstuhl, Germany, April. 2013.

[18] I. Konnov. *Counter Attack on Byzantine Generals.* Talk at the Seminar on Theoretical Problems in Programming, Moscow State University, Moscow, Russia, February. 2013.

[19] I. Konnov. *Who is Afraid of Model Checking Distributed Algorithms*. Talk at the PUMA/RiSE Seminar, Goldegg, Austria, September. 2012.

[20] I. Konnov. *An invariant-based approach to the verification of asynchronous parameterized networks*. Talk at the Concurrency Seminar, Computing Laboratory, Oxford University, Oxford, UK, February. 2011.

[21] I. Konnov. *Two Techniques of Parameterized Model Checking and Symmetry Reduction*. Talk at the RiSE Seminar, TU Vienna, Vienna, Austria, April. 2011.

[22] I. V. Konnov. *CheAPS: Parameterized Model Checking Tool*. Joint Workshop of Microsoft Research and Institute for System Programming Russian Academy of Sciences, Moscow, June 2009. 2009.