Tutorial on Parameterized Model Checking of Fault-Tolerant Distributed Algorithms*

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

Vienna University of Technology (TU Wien)

Abstract. Recently we introduced an abstraction method for parameterized model checking of threshold-based fault-tolerant distributed algorithms. We showed how to verify distributed algorithms without fixing the size of the system a priori. As is the case for many other published abstraction techniques, transferring the theory into a running tool is a challenge. It requires understanding of several verification techniques such as parametric data and counter abstraction, finite state model checking and abstraction refinement. In the resulting framework, all these techniques should interact in order to achieve a possibly high degree of automation. In this tutorial we use the core of a fault-tolerant distributed broadcasting algorithm as a case study to explain the concepts of our abstraction techniques, and discuss how they can be implemented.

1 Introduction

Distributed systems are crucial for today's computing applications, as they enable us to increase performance and reliability of computer systems, enable communication between users and computers that are geographically distributed, or allow us to provide computing services that can be accessed over the Internet. Distributed systems allow us to achieve that by the use of distributed algorithms. In fact, distributed algorithms have been studied extensively in the literature [62,11], and the central problems are well-understood. They differ from the fundamental problems in sequential (that is, non-distributed) systems. The central problems in distributed systems are posed by the inevitable uncertainty of any local view of the global system state, originating in unknown/varying processor speeds, communication delays, and failures. Pivotal services in distributed systems, such as mutual exclusion, routing, consensus, clock synchronization, leader election, atomic broadcasting, and replicated state machines, must hence be designed to cope with this uncertainty.

As we increasingly depend on the correct operation of distributed systems, the ability to cope with failures becomes particularly crucial. To do so, one actually has to address two problem areas. On the one hand, one has to design

[*] Some of the presented material has been published in [53,52]. 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.

algorithms that can deal with partial failure that is outside the control of a system designer. Typical examples are temporary disconnections of the network (e.g., due to mobility), power outages, bit-flips due to radiation in space, or hardware faults. On the other hand, we have to prevent, or rather find and remove, design faults, which are often termed as bugs. The former area of fault tolerance is classically addressed by means of replication and fault-tolerant distributed algorithms [62,11,25], while the latter is dealt with by rigorous software engineering methods such as model checking [31,12,47]. In order to maximize the reliability, one should deploy fault-tolerant distributed algorithms that have been verified.

We prefer model checking to verification using proof checkers such as PVS or Isabelle, as model checking promises a higher degree of automation, and still allows us to verify designs and implementation. Testing, on the other hand, can be completely automated and it allows us to validate large systems. However, there are still many research challenges in testing of distributed systems, and in general, testing suffers from being incomplete. Hence, model checking strikes a good balance between automatization and completeness. In verification of fault-tolerant distributed algorithms we are not looking for a push-button technology: First, as we will see below, distributed algorithms are naturally parameterized, and parameterized model checking is undecidable even for very simple systems [10,77]. Second, distributed algorithms are typically only given in natural language or pseudo code. Hence, in contrast to software model checking where the input is given as a program in, e.g., C, currently the input for the verification of distributed algorithms is not machine readable, and we require expert knowledge from the beginning. Finally, a method where the user (or rather the system designer) guides the model checking tool is acceptable if we can check automatically that the user input does not violate soundness.

Only very few fault-tolerant distributed algorithms have been automatically verified. We think that this is because many aspects of distributed algorithms still pose research challenges for model checking:

- The inherent concurrency and the uncertainty caused by partial failure lead to many sources of non-determinism. Thus, fault-tolerant distributed algorithms suffer from combinatorial explosion in the state-space and in the number of behaviors.
- For many applications, the size of the distributed system, that is, the number of participants is a priori unknown. Hence, the design and verification of distributed algorithms should work for all system sizes. That is, distributed systems are parameterized by construction.
- Distributed algorithms are typically only correct in certain environments, e.g., when there is only a certain fraction of the processes faulty, when the interleaving of steps is restricted, or when the message delays are bounded.
- Faults change the semantics of primitives (send, receive, FIFO, access object), classic primitives such as handshake may be impossible or impractical to implement.
- There is no commonly agreed-upon distributed computing model, but rather many variants, which differ in subtle details. Moreover, distributed algo-

rithms are usually described in pseudocode, typically using different (alas unspecified) pseudocode languages, which obfuscates the relation to the underlying computing model.

In this tutorial we discuss practical aspects of parameterized model checking of fault-tolerant distributed algorithms. We use Srikanth and Toueg's broadcasting primitive [76] as a case study, and discuss various aspects using encodings in Promela and Yices. The reader is thus expected to have basic knowledge of of Spin and Yices [2,5,49,38].

Srikanth and Toueg's broadcasting primitive is an example for threshold-based fault-tolerant algorithms, and our methods are tailored for this kind of distributed algorithms. We thus capture important mechanisms in distributed algorithms like waiting for messages from a majority of processes. Section 2 contains more detailed discussion on our motivations. We will discuss in detail the formalization of such algorithms in a parametric variant of Promela in Section 3. We then show in Section 4 how to use abstraction to reduce the parameterized model checking problem to a finite state model checking problem, and discuss how to deal with many practical issues that are due to abstraction. We show the efficiency of our method by experimental evaluation in Section 6.

2 Context

2.1 Parameterized Model Checking

In its original formulation [30], Model Checking was concerned with efficient procedures for the evaluation of a temporal logic specification φ over a finite Kripke structure K, i.e., decision procedures for $K \models \varphi$. Since K can be extremely large, a multitude of logic-based algorithmic methods including symbolic verification [64,18] and predicate abstraction [46] were developed to make this decidable problem tractable for practical applications. Finite-state models are, however, not always an adequate modeling formalism for software and hardware.

(i) Infinite-state models. Many programs and algorithms are naturally modeled by unbounded variables such as integers, lists, stacks etc. Modern model checkers are using predicate abstraction [46] in combination with SMT solvers to reduce an infinite-state model I to a finite state model $h(I)$ that is amenable to finite state model checking. The construction of h assures soundness, i.e., for a given specification logic such as ACTL^*, we can assure by construction that $h(I) \models \varphi$ implies $I \models \varphi$. The major drawback of abstraction is incompleteness: if $h(I) \not\models \varphi$ then it does in general not follow that $I \not\models \varphi$. (Note that ACTL^* is not closed under negation.) Counterexample-guided abstraction refinement (CEGAR) [27,13] addresses this problem by an adaptive procedure, which analyzes the abstract counterexample for $h(I) \not\models \varphi$ on $h(I)$ to find a concrete counterexample or obtain a better abstraction $h'(I)$. For abstraction to work in practice, it is crucial that the abstract domain from which h and h' are chosen is tailored to the problem class and possibly the specification. Abstraction thus is a semi-decision procedure whose usefulness has to be demonstrated by practical examples.

(ii) An orthogonal modeling and verification problem is parameterization: Many software and hardware artifacts are naturally represented by an infinite class of structures $\mathbf{K} = \{K_1, K_2, \dots\}$ rather than a single structure. Thus, the verification question is $\forall i.\, K_i \models \varphi$, where i is called the parameter. In the most important examples of this class, the parameter i is standing for the number of replications of a concurrent component, e.g., the number of processes in a distributed algorithm, or the number of caches in a cache coherence protocol. It is easy to see that even in the absence of concurrency, parameterized model checking is undecidable [10]; more interestingly, undecidability even holds for networks of constant size processes that are arranged in a ring and that exchange a single token [77,41]. Although several approaches have been made to identify decidable classes for parameterized verification [41,40,81], no decidable formalism has been found which covers a reasonably large class of interesting problems. The diversity of problem domains for parameterized verification and the difficulty of the problem gave rise to many approaches including regular model checking [6] and abstraction [70,28] — the method discussed here. The challenge in abstraction is to find an abstraction $h(\mathbf{K})$ such that $h(\mathbf{K}) \models \varphi$ implies $K_i \models \varphi$ for *all* i.

Most of the previous research on parameterized model checking focused on concurrent systems with $n + c$ processes where n is the parameter and c is a *constant*: n of the processes are *identical* copies; c processes represent the non-replicated part of the system, e.g., cache directories, shared memory, dispatcher processes etc. [45,50,65,28]. Most of the work on parameterized model checking considers only safety. Notable exceptions are [56,70] where several notions of fairness are considered in the context of abstraction to verify liveness.

2.2 Fault-tolerant Distributed Algorithms

In this tutorial we are not aiming at the most general approach towards parameterized model checking, but we are addressing a very specific problem in the field, namely, parameterized verification of fault-tolerant distributed algorithms (FTDA). This work is part of an interdisciplinary effort by the authors to develop a tool basis for the automated verification, and, in the long run, deployment of FTDAs [51,57]. FTDAs constitute a core topic of the distributed algorithms community with a rich body of results [62,11]. FTDAs are more difficult than the standard setting of parameterized model checking because *a certain number t of the n processes can be faulty.* In the case of e.g. Byzantine faults, this means that the faulty processes can send messages in an unrestricted manner. Importantly, the upper bound t for the faulty processes is also a parameter, and is essentially a fraction of n. The relationship between t and n is given by a *resilience condition*, e.g., $n > 3t$. Thus, one has to reason about all systems with $n - f$ non-faulty and f faulty processes, where $f \leq t$ and $n > 3t$.

From a more operational viewpoint, FTDAs typically consist of multiple processes that communicate by message passing over a completely connected communication graph. Since a sender can be faulty, a receiver cannot wait for a message from a specific sender process. Therefore, most FTDAs use counters to reason about their environment. If, for instance, a process receives a certain message m from more than t distinct processes, it can conclude that at least one of the senders is non-faulty. A large class of FTDAs [39,75,44,37,36] expresses these counting arguments using *threshold guards:*

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

Note that threshold guards generalize existential and universal guards [40], that is, 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 [51], existential and universal guards are not sufficient to capture advanced FTDAs.

2.3 The Formalization Problem

In the literature, the vast majority of distributed algorithms is described in pseudo code, for instance, [75,8,79]. The intended semantics of the pseudo code is folklore knowledge among the distributed computing community. Researchers who have been working in this community have intuitive understanding of keywords like “send”, “receive”, or “broadcast”. For instance, inside the community it is understood that there is a semantical difference between “send to all” and “broadcast” in the context of fault tolerance. Moreover, the constraints on the environment are given in a rather informal way. For instance, in the authenticated Byzantine model [39], it is assumed that faulty processes may behave arbitrarily. At the same time, it is assumed that there is some authentication service, which provides unbreakable digital signatures. In conclusion, it is thus assumed that faulty processes send any messages they like, *except* ones that look like messages sent by correct processes. However, inferring this kind of information about the behavior of faulty processes is a very intricate task.

At the bottom line, a close familiarity with the distributed algorithms community is required to adequately model a distributed algorithm in preparation of formal verification. When the essential conditions are hidden between the lines of a research paper, then one cannot be sure that the algorithm being verified is the one that is actually intended by the authors. With the current state of the art, we are thus forced to do *verification of a moving target.*

We conclude that there is need for a versatile specification language which can express distributed algorithms along with their environment. Such a language should be natural for distributed algorithms researchers, but provide unambiguous and clear semantics. Since distributed algorithms come with a wide range of different assumptions, the language has to be easily configurable to these situations. Unfortunately, most verification tools do not provide sufficiently expressive languages for this task. Thus, it is hard for researchers from the distributed computing community to use these tools out of the box. Although distributed algorithms are usually presented in a very compact form, the “language primitives” (of pseudo code) are used without consideration of implementation issues and computational complexity. For instance sets, and operations on sets are often used as they ease presentation of concepts to readers, although fixed size vectors would be sufficient to express the algorithm and more efficient to implement. Besides, it is not unusual to assume that any local computation on a node can be completed within one step. Another example is the handling of messages. For instance, how a process stores the messages that have been received in the past is usually not explained in detail. At the same time, quite complex operations are performed on this information.

2.4 Verified Fault-tolerant Distributed Algorithms

Several distributed algorithms have been formally verified in the literature. Typically, these papers have addressed specific algorithms in fixed computational models. There are roughly two lines of research. On the one hand, the semi-manual proofs conducted with proof assistants that typically involve an enormous amount of manual work by the user, and on the other hand automatic verification, e.g., via model checking. Among the work using proof assistants, Byzantine agreement in the synchronous case was considered in [61,73]. In the context of the heard-of model with message corruption [15] Isabelle proofs are given in [24]. For automatic verification, for instance, algorithms in the heard-of model were verified by (bounded) model checking [78]. Partial order reductions for a class of fault-tolerant distributed algorithms (with "quorum transitions") for fixed-size systems were introduced in [19]. A broadcasting algorithm for crash faults was considered in [43] in the context of regular model checking; however, the method has not been implemented so it is not clear how practical it is. In [9], the safety of synchronous broadcasting algorithms that tolerate crash or send omission faults has been verified. Another line of research studies decidability of model checking of distributed systems under different link semantics [7,22].

Model checking of fault-tolerant distributed algorithms is usually limited to small instances, i.e., to systems consisting of only few processes (e.g., 4 to 10). However, distributed algorithms are typically designed for parameterized systems, i.e., for systems of arbitrary size. The model checking community has created interesting results toward closing this gap, although it still remains a big research challenge. For specific cache coherence protocols, substantial research has been done on model checking safety properties for systems of arbitrary size, for instance, [65,26,68]. Since these protocols are usually described via message passing, they appear similar to asynchronous distributed algorithms. However, issues such as faulty components and liveness are not considered in the literature. The verification of large concurrent systems by reasoning about suitable small ones has also been considered [41,29,32,70].

3 Modeling Fault-tolerant Distributed Algorithms

3.1 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. [62,11]) 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 [62]. 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 non-determinism. In order to shed some light on the difficulties faced a distributed algorithm in the presence of faults, consider Byzantine faults [69], 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 [71].

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 that 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 just a 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 [39,67]), $t+1$ (to wait for a message from at least one correct process [76,39]), or $n-t$ (in the Byzantine case [76,8] 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.

3.2 Reliable broadcast and related specifications

The specifications considered in the field of fault tolerance differ from more classic fields, 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., in LTL, $\forall i.\ \mathsf{G}\,(\text{hungry}_i \rightarrow (\mathsf{F}\,\text{eating}_i))$, and thus to employ *indexed* temporal logics for specifications [21,28,29,41].

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 $\mathit{bcast}(i,m)$, where m is a unique message content. Processes may deliver a message by invoking $\mathit{accept}(i,m)$ for different process and message pairs (i,m). The goal is that all correct processes invoke $\mathit{accept}(i,m)$ 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 $\mathit{accept}(i,m)$ may not be invoked, unless i is faulty or i invoked $\mathit{bcast}(i,m)$. Our case study is to verify that the algorithm from [76] implements these primitives on top of point-to-point channels, in the presence of Byzantine faults. In [76] the specifications where given in natural language as follows:

- **(U) Unforgeability.** If correct process i does not broadcast (i,m), then no correct process ever accepts (i,m).
- **(C) Correctness.** If correct process i broadcasts (i,m), then every correct process accepts (i,m).
- **(R) Relay** If a correct process accepts (i,m), then every other correct process accepts (i,m).

In [76], 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 $\mathit{bcast}(i,m)$ that may occur, e.g., the cases where the invoking process is faulty or correct. As we focus on the core functionality, we do not model the broadcaster explicit. We observe that correct broadcasters will either send to all, or to no other correct processes. Hence, we model this by initial values V1 and V0 at correct processes that we use to model whether a process has received the message by the broadcaster or not, respectively. Then the precondition of correctness can be modeled that all correct processes initially have value V1, while the precondition of unforgeability that all correct processes initially have value V0. Depending on the initial state, we then have to check whether every/no correct process accepts (that is, changes the status to AC). 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) states that if some correct process accepts a message, then all (correct) processes accept the message, that is, $\mathsf{G}\,((\exists i.\ \mathrm{accept}_i) \rightarrow \mathsf{F}\,(\forall j.\ \mathrm{accept}_j))$.

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:

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

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

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

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.

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.

Observe that the specifications (C) and (R) are conditional liveness properties. Intuitively, a process has to find out that the condition is satisfied a run, and in distributed systems this is only possible by receiving messages. Specification (C) can thus only be achieved if some messages are received. Indeed, the algorithm in [76] is based on a property called *reliable communication* which ensures that every message sent by a correct process to a correct process is eventually received by the latter. Such properties can be expressed by justice requirements [70], which is a specific form of fairness. We will express justice as an $\mathsf{LTL} \setminus \mathsf{X}$ formula ψ over AP_D. Then, given 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 parameterized model checking problem is to verify whether

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

Algorithm 1 Core logic of the broadcasting algorithm from [76].

```
Code for processes $i$ if it is correct:
Variables
1: $v_i \in \{\textsc{false}, \textsc{true}\}$
2: $\text{accept}_i \in \{\textsc{false}, \textsc{true}\} \leftarrow \textsc{false}$

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

3.3 Threshold-guarded distributed algorithms in Promela

Algorithm 1 is our case study for which we also provide a complete Promela implementation later in Listing 3. To explain how we obtain this implementation, 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 or by abstraction.

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 $n > 3t \wedge f \leq t \wedge t > 0$. 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 [42]. (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 [62]. 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, the receive operation does never block the execution; 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 3.3.

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 in the concrete (non-parameterized) case. 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 $f \leq t$).

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 sender), 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 not to 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 size 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++; /*skip*/
   :: 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 to 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 */
 };
 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 $\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}$ describes a global state where messages are still in transit. It follows that a formula ψ defined by

$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}] \tag{RelComm}$$

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 on page 11) using channels. The implementations are direct consequences of the fault types description. Figure 2 shows how the impact of faults on processes following the algorithm can be implemented in the shared

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

Listing 1:

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

Listing 2:

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

memory implementation of message passing. Note that in contrast to 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:

$$\mathsf{G}\,\mathsf{F}\,\neg\,[\exists i.\ \mathit{rcvd}_i + f < \mathit{nsnt}]$$

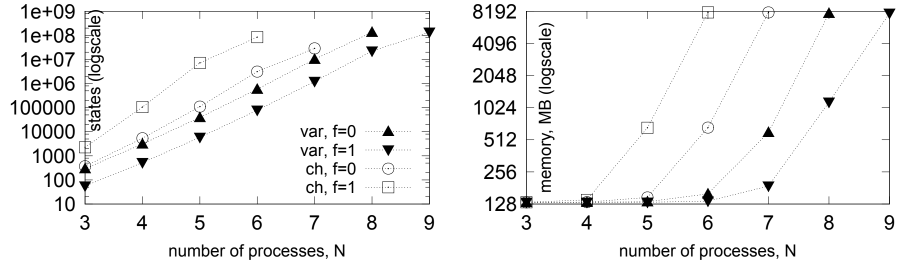


Fig. 3: Visited states (left) and memory usage (right) when modeling message passing with channels (ch) or shared variables (var). The faults are in effect only when $f > 0$. Ran with `SAFETY`, `COLLAPSE`, `COMP`, and 8GB of memory.

By similar adaptations one models, e.g., corrupted communication (e.g., due to faulty links) [72], or hybrid fault models [16] that contain different fault scenarios.

Comparing Promela encodings: Channels vs. Shared Variables Figure 3 compares the number of states and memory consumption when modeling message passing using both solutions. We ran Spin to perform exhaustive state enumeration on the encoding of our case study algorithm in Listing 3. 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 to the use the shared memory encoding based on *nsnt* variables. In addition, we have seen in the previous section that this encoding is very natural for defining abstractions.

Encoding the control flow. Recall Algorithm 1 on page 10, 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 Listing 3 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 [52]. 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`.

```
symbolic int N, T, F;  /* parameters  */
/* the resilience condition */
assume(N > 3 * T && T >= 1 && 0 <= F && F <= T);
int nsnt;  /* number of echoes sent by correct processes */
/* quantified atomic propositions */
atomic prec_unforg = all(STBcast:sv == V0);
atomic prec_corr = all(STBcast:sv == V1);
atomic prec_init = all(STBcast@step);
atomic ex_acc = some(STBcast:sv == AC);
atomic all_acc = all(STBcast:sv == AC);
atomic in_transit = some(STBcast:nrcvd < nsnt);

active[N - F] proctype STBcast() {
  byte sv, next_sv;          /* status of the algorithm */
  int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes received */
  if  /* initialize */
    :: sv = V0; /* v_i = FALSE */
    :: sv = V1; /* v_i = TRUE */
  fi;
step: atomic { /* an indivisible step */
    if /* receive one more echo (up to nsnt + F) */
      :: (nrcvd < nsnt + F) -> next_nrcvd = nrcvd + 1;
      :: next_nrcvd = nrcvd; /* or nothing */
    fi;
    if /* compute */
      :: (next_nrcvd >= N - T) ->
        next_sv = AC; /* accept_i = TRUE */
      :: (next_nrcvd < N - T
          && (sv == V1 || next_nrcvd >= T + 1)) ->
        next_sv = SE; /* remember that <echo> is sent */
      :: else -> next_sv = sv; /* keep the status */
    fi;
    if /* send */
      :: (sv == V0 || sv == V1)
          && (next_sv == SE || next_sv == AC) ->
        nsnt++; /* send <echo>  */
      :: else;  /* send nothing */
    fi;
    /* update local variables and reset scratch variables */
    sv = next_sv; nrcvd = next_nrcvd;
    next_sv = 0; next_nrcvd = 0;
    } goto step;
}
ltl fairness { []<>(!in_transit) } /* fairness -> formula */
/* LTL-X formulas */
ltl relay { [](ex_acc -> <>all_acc) }
ltl corr { []((prec_init && prec_corr) -> <>(ex_acc)) }
ltl unforg { []((prec_init && prec_unforg) -> []!ex_acc) }
```

Listing 3: Encoding of Algorithm 1 in parametric PROMELA.

In the encoding in Listing 3, 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 $\langle echo \rangle$ before", "if v_i", and "$accept_i \leftarrow$ TRUE". We capture all possible control states in a finite set SV. For instance, for Algorithm 1 one can collect the set $SV = \{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$, where:

- V0 corresponds to $v_i =$ FALSE, $\mathrm{accept}_i =$ FALSE and $\langle \mathrm{echo} \rangle$ is not sent.
- V1 corresponds to $v_i =$ TRUE, $\mathrm{accept}_i =$ FALSE and $\langle \mathrm{echo} \rangle$ is not sent.
- SE corresponds to the case $\mathrm{accept}_i =$ FALSE and $\langle \mathrm{echo} \rangle$ been sent. Observe that once a process has sent $\langle \mathrm{echo} \rangle$, its value of v_i does not interfere anymore with the subsequent control flow.
- AC corresponds to the case $\mathrm{accept}_i =$ TRUE and $\langle \mathrm{echo} \rangle$ 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 $SV_0 = \{\mathrm{V0}, \mathrm{V1}\}$ of initial control states.

Formalization This paper is a hands-on tutorial on parameterized model checking. So we will use PROMELA to explain our methods in the following sections. Note that we presented the theoretical foundations of these methods in [52]. In this paper we will restrict ourselves to introduce some definitions that make it easier to discuss the central ideas of our abstraction.

In the code we use variables of different roles: we have parameters (e.g., n, t, and f), local variables (*rcvd*) and shared variables (*nsnt*). We will denote by $\mathit{\Pi}$, $\mathit{\Lambda}$, and $\mathit{\Gamma}$ the sets of parameters, local variables, and shared variables, respectively. All these variables range over a *domain* D that is totally ordered and has the operations of addition and subtraction, e.g., the set of natural numbers $\mathbb{N}_0$. We have discussed above that fault-tolerant distributed algorithms can tolerate only certain fractions of processes to be faulty. We capture this using the *resilience condition* RC that is a predicate over the values of variables in $\mathit{\Pi}$. In our example, $\mathit{\Pi} = \{n, t, f\}$, 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 $\mathbf{P}_{RC} = \{\mathbf{p} \in D^{|\mathit{\Pi}|} \mid RC(\mathbf{p})\}$.

As we have seen, a system instance is a parallel composition of identical processes. 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 $N \colon \mathbf{P}_{RC} \to \mathbb{N}$, for instance, in our example we model only correct processes explicitly, and so we use $n - f$ for $N(n, t, f)$.

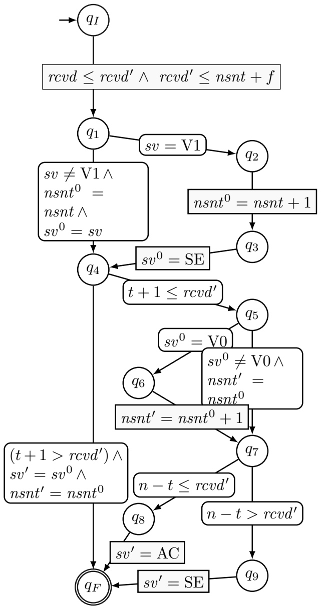


Fig. 4: CFA of our case study for Byzantine faults.

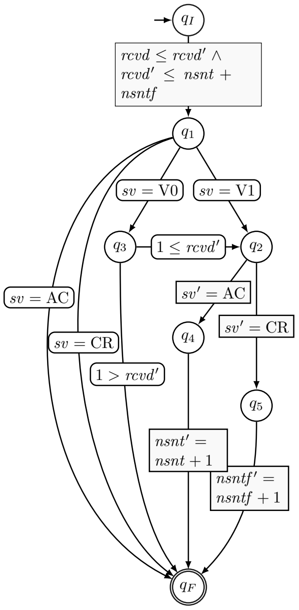


Fig. 5: CFA of FTDA from [43] (if x' is not assigned, then $x' = x$).

To model how the system evolves, that is, to model a step of a process, we use control flow automata (CFA). They formalize fault-tolerant distributed algorithms. Figure 4 gives the CFA of our case study algorithm. 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).

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.

Recall that 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.

4 Abstraction

In this section we demonstrate how one can apply various abstractions to reduce a parameterized model checking problem to a finite-state model checking prob-

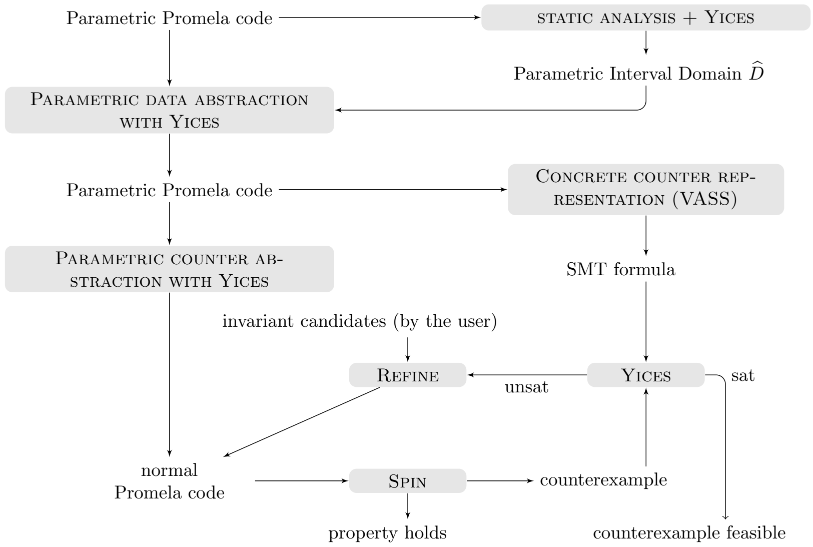

Parametric Promela code
static analysis + Yices
Parametric Interval Domain $\widehat{D}$
Parametric data abstraction with Yices
Parametric Promela code
Concrete counter representation (VASS)
Parametric counter abstraction with Yices
SMT formula
invariant candidates (by the user)
Refine
unsat
Yices
sat
normal Promela code
Spin
counterexample
property holds
counterexample feasible

Fig. 6: The abstraction scheme

lem. An overview is given in Figure 6. We show how the abstraction works on the code level, that is, how the parametric Promela program constructed in Section 3 is translated to a program in standard Promela. Since we are interested in parameterized model checking, we need to ensure that the specifications are satisfied in concrete systems of all sizes. Hence, we need an abstract system that contains all behaviors that are experienced in concrete systems. Consequently, we use existential abstraction which ensures that if there exists a concrete system and a concrete run in that system, this run is mapped to a run in the abstract system. In that way, if there exists a system in which a specification is violated, the specification will also be violated in the abstract system. In other words, if we can verify a specification in the abstract system, the specification holds in *all* concrete systems; we say the verification method is sound. The formal exposition can be found in [52].

Usually abstractions introduce new behavior that is not present in the original system. Thus, a finite-state model checker might find a spurious run, that is, one that none of the concrete systems with fixed parameters can replay. In order to discard such runs, one applies abstraction refinement techniques [27].

In what follows, we demonstrate three levels of abstraction: parametric interval data abstraction, parametric interval counter abstraction, and parametric interval data abstraction of the local state space. The first two abstractions are used for reducing a parameterized problem to a finite-state one, while the third abstraction helps us to detect spurious counterexamples.

Throughout this chapter we are using the core part of asynchronous reliable broadcast by Srikanth&Toueg as our running example. Its encoding in parametric PROMELA is given in Listing 3. Our final goal is to obtain a PROMELA program that we can verify in SPIN.

4.1 Parametric Interval Data Abstraction

Let us have a look at the code on Listing 3. The process prototype `STBcast` refers to two kinds of variables, each of them having a special role:

- *Bounded variables.* These are local variables that range over a finite domain, the size of which is independent of the parameters. In our example, the variable of this kind are `sv` and `next_sv`.
- *Unbounded variables.* These are the variables that range over an unbounded domain. They may be local or shared. In our example, the variables `nrcvd`, `next_nrcvd`, and `nsnt` are unbounded. It might happen that the variables become bounded, when one fixes the parameters, as it is the case in our example with `nsnt` $\leq n - f$. However, we need a finite representation independent of the parameters, that is, the bounds on the variable values must be independent of the parameter values.

We can partition the variables into the sets B (bounded) and U (unbounded) by performing value analysis on the process body. Intuitively, one can imagine that the analysis iteratively computes the set B of variables that are assigned their values only using the following two kinds of statements:

- An assignment that copies a constant expression to a variable;
- An assignment that copies the value of another variable, which already belongs to B.

The variables outside of B, e.g., those that are incremented in the code, belong to U. As this can be done by a simple implementation of abstract interpretation [33], we omit the details here.

The data abstraction that we are going to explain below deals with unbounded variables by turning the operations over unbounded domains into operations over finite domains. The threshold-based fault-tolerant distributed algorithms give us a natural source of abstract values, namely, the threshold expressions. In our example, the variable `next_nrcvd` is compared against thresholds $t + 1$ and $n - t$. Thus, it appears natural to forget about concrete values of `next_nrcvd`. As a first try, we may replace the expressions that involve `next_nrcvd` with the expressions over the two predicates: `p1_next_nrcvd` $\equiv x < t + 1$ and `p2_next_nrcvd` $\equiv x < n - t$. Then, the following code is an abstraction of the computation block in lines (25)–(32) of Listing 3:

```
if /* compute */
  :: (!p2_next_nrcvd) -> next_sv = AC;
  :: (!p2_next_nrcvd && (sv == V1 || !p1_next_nrcvd)) ->
    next_sv = SE;
  :: else -> next_sv = sv;
fi;
```

Listing 4: Predicate abstraction of the computation block

In principle, we could use this kind of predicate abstraction for our purposes. However, we have seen that our modeling involves considerable amounts of arithmetics, e.g., code line 22 in our example contains comparison of two variables as well increasing the value of a variable. Such notions are not naturally expressed in terms of predicate abstraction. Rather, we introduce a parametric interval abstraction PIA, which is based on an abstract domain that represents intervals, whose boundaries are expressions to which variables are compared to; e.g., $t+1$ and $n-t$. We then use an SMT solver to abstract expressions, e.g., comparisons.

Hence, instead of using several predicates, we can replace the concrete domain of every variable $x \in U$ with the abstract domain $\{I_0, I_{t+1}, I_{n-t}\}$. For reasons that are motivated by the counter abstraction — to be introduced later in Section 4.2 — we have to distinguish value 0 from a positive value. Thus, we are extending the domain with the threshold "1", that is, $\widehat{D} = \{I_0, I_1, I_{t+1}, I_{n-t}\}$.

The semantics of the abstract domain is as follows. We introduce an abstract version of x, denoted by $\hat{x}$; its values (from $\widehat{D}$) relate to the concrete values of x as follows: $\hat{x} = I_0$ iff $x \in [0;1[$ and $\hat{x} = I_1$ iff $x \in [1;t+1[$ and $\hat{x} = I_{t+1}$ iff $x \in [t+1;n-t[$ and $\hat{x} = I_{n-t}$ iff $x \in [n-t;\infty[$. Having defined the abstract domain, we translate the computation block in lines (25)–(32) of Listing 3 as follows (we discuss below how the translation is done automatically):

```
if /* compute */
 :: next_nrcvd == $I_{n-t}$ -> next_sv = AC;
 :: (next_nrcvd == $I_0$ || next_nrcvd == $I_1$ || next_nrcvd== $I_{t+1}$)
   && (sv == V1 || (next_nrcvd == $I_{t+1}$ || next_nrcvd == $I_{n-t}$))
   -> next_sv = SE;
 :: else -> next_sv = sv;
fi;
```

Listing 5: Parametric interval abstraction of the computation block

The abstraction of the receive block (cf. lines 21–24 of Listing 3) involves the assignment `next_nrcvd = nrcvd + 1` that becomes a *non-deterministic choice* of the abstract value of `next_nrcvd` based on the abstract value of `nrcvd`. Intuitively, `next_nrcvd` could be in the same interval as `nrcvd` or in the interval above. In the following, we provide the abstraction of lines 21–24, we will discuss later how this abstraction can be computed using an SMT solver.

```
  if /* receive */
    :: (/* abstraction of (next_nrcvd < nsnt + F) */) ->
    if :: nrcvd == $I_0$ -> next_nrcvd = $I_1$;
       :: nrcvd == $I_1$ -> next_nrcvd = $I_1$;
       :: nrcvd == $I_1$ -> next_nrcvd = $I_{t+1}$;
       :: nrcvd == $I_{t+1}$ -> next_nrcvd = $I_{t+1}$;
       :: nrcvd == $I_{t+1}$ -> next_nrcvd = $I_{n-t}$;
       :: nrcvd == $I_{n-t}$ -> next_nrcvd = $I_{n-t}$;
    fi;
    :: next_nrcvd = nrcvd;
  fi;
```

Listing 6: Parametric interval abstraction of the receive block

There are several interesting consequences of transforming the receive block as above. First, due to our resilience condition (which ensures that intervals do not overlap) for every value of `nrcvd` there are at most two values that can be assigned to `next_nrcvd`. For instance, if `nrcvd` equals I_{t+1}, then `next_nrcvd` becomes either I_{t+1}, or I_{n-t}. Second, due to non-determinism, the assignment is not anymore guaranteed to reach any value, e.g., `next_nrcvd` might be always assigned value I_1.

Formalization In the following, we explain the mathematics behind the idea of parametric interval abstraction, and the intuition why it is precise for specific expressions. To do so, we start with some preliminary definitions, which allow us to define parameterized abstraction functions and the corresponding concretization functions. We then make precise what it means to be an existential abstraction and derive questions for the SMT solver whose response will provide us with the abstractions of the PROMELA code discussed above.

Consider the arithmetic expressions over constants and parameters that are used in comparisons against unbounded variables, e.g., `next_nrcvd <= t+1`. From this we get expressions, e.g., `t+1` to which variables are compared. Let set $\mathcal{T}$ include all such expressions as well as the constants 0 and 1, and $\mu + 1$ be the cardinality of $\mathcal{T}$. We call the elements of $\mathcal{T}$ *thresholds*, and name them as as $e_0, e_1, \ldots, e_\mu$; with e_0 corresponding to the constant 0, and e_1 corresponding to 1.[1] Note that by evaluating threshold expressions for fixed parameters, we obtain a constant value of the threshold. Given a parameter evaluation $\mathbf{p}$ from $\mathbf{P}_{RC}$, we will denote by $e_i(\mathbf{p})$ the value of the ith threshold under $\mathbf{p}$. Given $\mathcal{T}$, we define the domain of parametric intervals as: $\widehat{D} = \{I_j \mid 0 \leq j \leq \mu\}$. Observe that in our running example we actually write $\widehat{D} = \{I_0, I_1, I_{t+1}, I_{n-t}\}$, to make it more intuitive. This is an abuse of notation, and following the above definition strictly, one has to write the domain as $\{I_0, I_1, I_2, I_3\}$.

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

Definition 1. *The finite set $\mathcal{T}$ is uniformly ordered if for all $\mathbf{p} \in \mathbf{P}_{RC}$, and all $e_j(\mathbf{p})$ and $e_k(\mathbf{p})$ in $\mathcal{T}$ with $0 \leq j < k \leq \mu$, it holds that $e_j(\mathbf{p}) < e_k(\mathbf{p})$.*

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.

[1] We add 0 and 1 explicitely, because we will later see that these values precisely capture an existential quantifier, similar to [70]. However, in our setting, the abstract domain that distinguishes between 0, 1, and *more* [70] is too coarse to track whether variables have surpassed certain thresholds.

Definition 1 allows us to properly define the *parameterized abstraction function* $\alpha_{\mathbf{p}} \colon D \to \widehat{D}$ and the *parameterized concretization function* $\gamma_{\mathbf{p}} \colon \widehat{D} \to 2^D$.

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

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

From $e_0(\mathbf{p}) = 0$ and $e_1(\mathbf{p}) = 1$, 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 1 one immediately obtains:

Proposition 1. *For all $\mathbf{p}$ in $\mathbf{P}_{RC}$, for all a in D, it holds that $a \in \gamma_{\mathbf{p}}(\alpha_{\mathbf{p}}(a))$.*

Definition 2. *We define comparison between parametric intervals I_k and I_ℓ as $I_k \le I_\ell$ iff $k \le \ell$.*

Compared to the predicate abstraction approach initially discussed, Definition 2 is very naturally written in our parametric interval abstraction, and we can use it in the following. In fact, the central property of our abstract domain is that it allows to abstract comparisons against thresholds in a precise way. That is, we can abstract formulas of the form $e_j(\mathbf{p}) \le x_1$ by $I_j \le \hat{x}_1$ and $e_j(\mathbf{p}) > x_1$ by $I_j > \hat{x}_1$. This abstraction is precise in the following sense.

Proposition 2. *For all $\mathbf{p} \in \mathbf{P}_{RC}$ and all $a \in D$:*
$e_j(\mathbf{p}) \le a$ iff $I_j \le \alpha_{\mathbf{p}}(a)$, and $e_j(\mathbf{p}) > a$ iff $I_j > \alpha_{\mathbf{p}}(a)$.

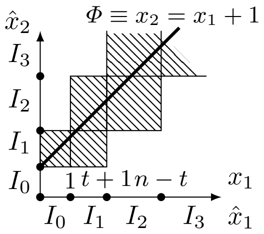
We now discuss what is necessary to construct an existential abstraction of an expression that involves comparisons against unbounded variables using an SMT solver. Let Φ be a formula that corresponds to such an expression. We introduce notation for sets of vectors satisfying Φ. Formula Φ has two kinds of free variables: parameter variables from Π and data variables from $\Lambda \cup \Gamma$. Let $\boldsymbol{x}^p$ be a vector of parameter variables $(x_1^p, \dots, x_{|\Pi|}^p)$ and $\boldsymbol{x}^d$ be a vector of variables $(x_1^d, \dots, x_k^d)$ over D^k. Given a k-dimensional vector $\boldsymbol{d}$ of values from D, by

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

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

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

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 $\hat{\boldsymbol{x}} = (\hat{x}_1, \dots, \hat{x}_k)$ over $\widehat{D}^k$ such that $\{\hat{\boldsymbol{d}} \in \widehat{D}^k \mid \hat{\boldsymbol{x}} = \hat{\boldsymbol{d}} \models \hat{\Phi}\} \supseteq ||\Phi||_{\exists}$. See Figure 7 for an example.



$$
\begin{aligned}
\hat{\Phi} \;\equiv\; & \hat{x}_1 = I_0 \wedge \hat{x}_2 = I_1 \\
\vee\, & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_1 \\
\vee\, & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_2 \\
\vee\, & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_2 \\
\vee\, & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_3 \\
\vee\, & \hat{x}_1 = I_3 \wedge \hat{x}_2 = I_3
\end{aligned}
$$

Fig. 7: 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 $\hat{\Phi}$.

Computing the abstractions. So far, we have seen the abstraction examples and the formal machinery in the form of existential abstraction $||\Phi||_\exists$. Now we show how to compute the abstractions using an SMT solver. We are using the input language of Yices [38], but this choice is not essential. Any other solver that supports linear arithmetics over integers, e.g., Z3 [35], should be sufficient for our purposes. We start with declaring the parameters and the resilience condition:

```
(define n :: int)
(define t :: int)
(define f :: int)
(assert (and (> n 3) (>= f 0)
             (>= t 1) (<= f t) (> n (* 3 t))))
```

Listing 7: The parameters and the resilience condition in Yices

Assume that we want to compute the existential abstraction of an expression similar to one found in line 22, that is,

$$\Phi_1 \equiv a < b + f.$$

According to the definition of $||\Phi_1||_\exists$, we have to enumerate all abstract values of a and b and check, whether there exist a valuation of the parameters n, t, and f and a concretization $\gamma_{n,t,f}$ of the abstract values that satisfies Φ_1. In the case of Φ_1 this boils down to finding all the abstract pairs $(\hat{a}, \hat{b}) \in \widehat{D} \times \widehat{D}$ satisfying the formula:

$$\exists a, b : \alpha_{n,t,f}(a) = \hat{a} \wedge \alpha_{n,t,f}(b) = \hat{b} \wedge a < b + f \tag{1}$$

Given $\hat{a}$ and $\hat{b}$, Formula (1) can be encoded as a satisfiability problem in linear integer arithmetics. For instance, if $\hat{a} = I_1$ and $\hat{b} = I_0$, then we encode Formula (1) as follows:

```
6  (push)                    ;; store the context for the future use
7  (define a :: int)
8  (define b :: int)
9  (assert (and (>= a 1) (< a (+ t 1)))) ;; $\alpha_{n,t,f}(a) = I_1$
10 (assert (and (>= b 0) (< b 1)))       ;; $\alpha_{n,t,f}(b) = I_0$
11 (assert (< a (+ b f)))                ;; $\Phi_1$
12 (check)                               ;; is satisfiable?
13 (pop)                     ;; restore the previously saved context
```

Listing 8: Are there a and b with $a < b + f$, $\alpha_{n,t,f}(a) = I_1$, and $\alpha_{n,t,f}(b) = I_0$?

When we execute lines (1)–(13) of Listing 8 in Yices, we receive `sat` on the output, that is, formula 1 is valid for the values $\hat{a} = I_1$ and $\hat{b} = I_0$ and $(I_1, I_0) \in ||a < b + f||_\exists$. To see concrete values of a, b, n, t, and f satisfying lines (1)–(13), we issue the following command:

```
14 (set-evidence! true)
15 ;; copy lines (1) - (13) here
```

Yices provides us with the following model:

```
(= n 7)
(= f 2)
(= t 2)
(= a 1)
(= b 0)
```

By enumerating all values from $\widehat{D} \times \widehat{D}$, we obtain the following abstraction of $a < b + f$ (this is an abstraction of line (22) in Listing 3):

```
   a == $I_{n-t}$ && b == $I_{n-t}$ || a == $I_{t+1}$ && b == $I_{n-t}$
|| a == $I_1$ && b == $I_{n-t}$ || a == $I_0$ && b == $I_{n-t}$
|| a == $I_{n-t}$ && b == $I_{t+1}$ || a == $I_{t+1}$ && b == $I_{t+1}$
|| a == $I_1$ && b == $I_{t+1}$ || a == $I_0$ && b == $I_{t+1}$
|| a == $I_{t+1}$ && b == $I_1$ || a == $I_1$ && b == $I_1$
|| a == $I_0$ && b == $I_1$ || a == $I_1$ && b == $I_0$ || a == $I_0$ && b== $I_0$
```

Listing 9: Parametric interval abstraction of $a < b + f$

By applying the same principle to all expressions in Listing 3, we abstract the process code. As the abstract code is too verbose, we do not give it here. It can be obtained by running the tool on our benchmarks [1], as described in Section 6.1.

Specifications. As we have seen in Section 3.2, we use only specifications that compare status variable sv against a value from SV. For instance, the unforgeability property **U** (cf. p. 9) is referring to atomic proposition $[\forall i.\ sv_i \neq \mathrm{V1}]$. Interval data abstraction does neither affect the domain of sv, nor does it change expressions over sv. Thus, we do not have to change the specifications when applying the data abstraction.

However, the specifications are verified under justice constraints, e.g., the reliable communication constraint (cf. RelComm on p. 13): $\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_i < nsnt]$.

Our goal is that the abstraction preserves fair (i.e., just) runs, that is, if each state of a just run is abstracted, then the resulting sequence of abstract states is a just run of the abstract system. Intuitively, when we verify a property that holds on all *abstract* just runs, then we conclude that the property also holds on all *concrete* just runs. In fact, we apply existential abstraction to the formulas that capture just states, e.g., we transform the expression $\neg\,[\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}]$ using existential abstraction $||\neg\,[\exists i.\ \mathit{rcvd}_i < \mathit{nsnt}]\,||_\exists$.

Let ψ be a propositional formula that describes just states, and $[\![\psi]\!]_{\mathbf{p}}$ be the set of states that satisfy ψ in the concrete system with the parameter values $\mathbf{p} \in \mathbf{P}_{RC}$. Then, by the definition of existential abstraction, for all $\mathbf{p} \in \mathbf{P}_{RC}$, it holds that $[\![\psi]\!]_{\mathbf{p}}$ is contained in the concretization of $||\psi||_\exists$. This property ensures justice preservation. In fact, we implemented a more general approach that involves *existential* and *universal* abstractions, but we are not going into details here. The interested reader can find formal frameworks in [56,74].

Remark on the precision. One may argue that domain $\widehat{D}$ is too imprecise and it might be helpful to add more elements to $\widehat{D}$. By Proposition 2, however, the domain gives us a *precise* abstraction of the comparisons against the thresholds. Thus, we do not lose precision when abstracting the expressions like $\mathit{next\_nrcvd} < t{+}1$ and $\mathit{next\_nrcvd} \geq n{-}t$, and we cannot benefit from enriching the abstract domain $\widehat{D}$ with expressions different from the thresholds.

4.2 Parametric Interval Counter abstraction

In the previous section we abstracted a process that is parameterized into a finite-state process. In this section we turn a system parameterized in the number of finite-state processes into a one-process system with finitely many states. First, we fix parameters $\mathbf{p}$ and show how one can convert a system of $N(\mathbf{p})$ processes into a one-process system by using a counting argument.

Counter representation. The structure of the PROMELA program after applying the data abstraction from Section 4.1 looks as follows:

```
int nsnt: 2 = 0;          /* 0 ↦ I_0 , 1 ↦ I_1 , 2 ↦ I_{t+1} , 3 ↦ I_{n−t} */
active[n - f] proctype Proc() {
  int pc: 2 = 0;          /* 0 ↦ V0 , 1 ↦ V1 , 2 ↦ SE , 3 ↦ AC */
  int nrcvd: 2 = 0;       /* 0 ↦ I_0 , 1 ↦ I_1 , 2 ↦ I_{t+1} , 3 ↦ I_{n−t} */
  int next_pc: 2 = 0, next_nrcvd: 2 = 0;
  if  :: pc = 0; /* V0 */
      :: pc = 1; /* V1 */
  fi;
loop: atomic {
    /* receive */
    /* compute */
    /* send */ }
  goto loop;
}
```

Listing 10: Process structure after data abstraction

Observe that a system consists of $N(\mathbf{p})$ identical processes. We may thus change the representation of a global state: Instead of storing which process is in which local state, we just count for each local state how many processes are in it. We have seen in the previous section that after the PIA data abstraction, processes have a fixed number of states. Hence, we can use a fixed number of counters. To this end, we introduce a global array of counters k that keeps the number of processes in every potential local state. We denote by L the set of local states and by L_0 the set of initial local states. In order to map the local states to array indices, we define a bijection: $h\colon L \to \{0, \dots, |L| - 1\}$.

In our example, we have 16 potential local states, i.e., $L_{ST} = \{(pc, nrcvd) \mid pc \in \{V0, V1, SE, AC\}, nrcvd \in \widehat{D}\}$. In our Promela encoding, the elements of $\widehat{D}$ and SV are represented as integers; we represent this encoding by the function $val\colon \widehat{D} \cup SV \to \{0, 1, 2, 3\}$ so that no two elements of $\widehat{D}$ are mapped to the same number and no two elements of SV are mapped to the same number. We allocate 16 elements for k and define the mapping $h_{ST} : L_{ST} \to \{0, \dots, |L_{ST}|-1\}$ as $h_{ST}((pc, nrcvd)) = 4 \cdot val(pc) + val(nrcvd)$. Then $k[h_{ST}(\ell)]$ stores how many processes are in local state ℓ. Thus, a global state is given by the array k, and the global variable $nsnt$.

It remains to define the transition relation. As we have to capture interleaving semantics, intuitively, if a process is in local state ℓ and goes to a different local state ℓ', then $k[h_{ST}(\ell)]$ must be decreased by 1 and $k[h_{ST}(\ell')]$ must be increased by 1. To do so in our encoding, we first *select* a state ℓ to move away from, perform a step as above, that is, calculate the successor state ℓ', and finally update the counters. Thus, the template of the counter representation looks as follows (we will discuss the select, receive, etc. blocks below):

```
int k[16]; /* number of processes in every local state */
int nsnt: 2 = 0;
active[1] proctype CtrAbs() {
  int pc: 2 = 0, nrcvd: 2 = 0;
  int next_pc: 2 = 0, next_nrcvd: 2 = 0;
  /* init */
loop: /* select */
    /* receive */
    /* compute */
    /* send */
    /* update counters */
  goto loop;
}
```

Listing 11: Process structure of counter representation

The blocks *receive*, *compute*, and *send* stay the same, as they were in Section 4.1. The new blocks have the following semantics: In *init*, an initial combination of counters is chosen such that $\sum_{\ell \in L_0} k[\ell] = N(\mathbf{p})$ and $\sum_{\ell \in L \setminus L_0} k[\ell] = 0$. In *select*, a local state ℓ with $k[\ell] \neq 0$ is non-deterministically chosen; In *update counters*, the counters of ℓ and a successor of ℓ are decremented and incremented respectively.

We now consider the blocks in detail and start with *init*. Each of $n - f$ processes start in one of the two initial states: $(V0, I_0)$ with $h_{ST}((V0, I_0)) = 0$ and $(V1, I_0)$ with $h_{ST}((V1, I_0)) = 3$. Thus, the initial block non-deterministically chooses the values for the counters $k[0]$ and $k[3]$, so that $k[0] + k[3] = n - f$ and all the other indices are set to zero. The following code fragment encodes this non-deterministic choice. Observe that the number of choices needed is $n - f + 1$, so the length of this code must depend on the choices of these parameters. We will get rid of this requirement in the counter abstraction below.

```
if /* 0 ↦ (pc = V0, nrcvd = I_0) ; 3 ↦ (pc = V1, nrcvd = I_0) */
  :: k[0] = n - f; k[3] = 0;
  :: k[0] = n - f - 1; k[3] = 1;
  ...
  :: k[0] = 0; k[3] = n - f;
fi;
```

In the *select* block we pick non-deterministically a non-zero counter $k[\ell]$ and set *pc* and *nrcvd* so that $h_{ST}(pc, nrcvd) = \ell$. Again, here is a small fragment of the code:

```
if
  :: k[0] != 0  -> pc = 0 /* V0 */; nrcvd = 0 /* I_0 */;
  :: k[1] != 0  -> pc = 0 /* V0 */; nrcvd = 1 /* I_1 */;
  ...
  :: k[15] != 0 -> pc = 3 /* AC */; nrcvd = 3 /* I_{n-t} */;
fi;
```

Finally, as the *compute* block assigns new values to *next_pc* and *next_nrcvd*, which correspond to the successor state of $(pc, nrcvd)$, we update the counters to reflect the fact that one process moved from state $(pc, nrcvd)$ to state $(next\_pc, next\_nrcvd)$:

```
if
  :: pc != next_pc || nrcvd != next_nrcvd ->
      k[4 * pc + nrcvd]--; k[4 * next_pc + next_nrcvd]++;
  :: else; /* do not update the counters */
fi;
```

This representation might look inefficient in comparison to the one with explicit processes; e.g., SPIN cannot use partial order reduction on this representation. However, this representation is only an intermediate step.

Specifications. In the original presentation of the system it is obvious how global states are linked with atomic propositions of the form $[\exists i.\ \Phi(i)]$ and $[\forall i.\ \Phi(i)]$; a process i must satisfy $\Phi(i)$ or all processes must do so, respectively. In the counter representation we do not "have" processes in the system anymore, and we have to understand which states to label with our atomic propositions.

In the counter representation, we exploit the fact that our properties are all quantified, which naturally translates to statements about counters: Let $[\![\Phi]\!]$ be the set of local states that satisfy Φ. In our example we are interested in the

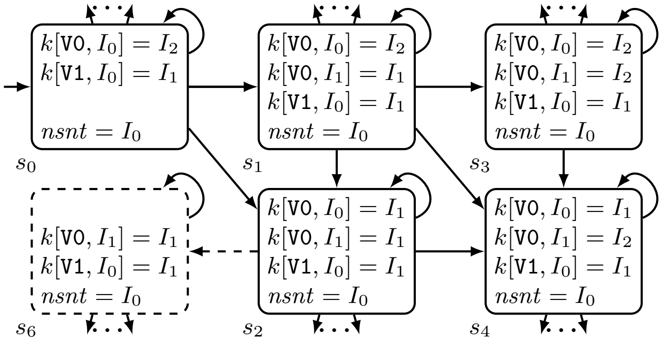

$k[\mathtt{V0}, I_0] = I_2$
$k[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_0
$k[\mathtt{V0}, I_0] = I_2$
$k[\mathtt{V0}, I_1] = I_1$
$k[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_1
$k[\mathtt{V0}, I_0] = I_2$
$k[\mathtt{V0}, I_1] = I_2$
$k[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_3
$k[\mathtt{V0}, I_1] = I_1$
$k[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_6
$k[\mathtt{V0}, I_0] = I_1$
$k[\mathtt{V0}, I_1] = I_1$
$k[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_2
$k[\mathtt{V0}, I_0] = I_1$
$k[\mathtt{V0}, I_1] = I_2$
$k[\mathtt{V1}, I_0] = I_1$
$nsnt = I_0$
s_4
...

Fig. 8: A small part of the transition system obtained by counter abstraction.

local states that satisfy $sv = \mathrm{AC}$, as it appears in our specifications. There are several such states (not all reachable, though) depending on the different values of *nsnt*. Then, a global state satisfies $[\exists i.\ \Phi(i)]$ if $\bigvee_{\ell \in [\![\Phi]\!]} k[\ell] \neq 0$. Similarly, a global state satisfies $[\forall i.\ \Phi(i)]$ if $\bigwedge_{\ell \notin [\![\Phi]\!]} k[\ell] = 0$.

As we are dealing with counters, instead of using disjunctions and conjunctions, we could also use sums to evaluate quantifiers: the universal quantifier could also be expressed as $\sum_{\ell \in [\![\Phi]\!]} k[\ell] = N(\mathbf{p})$. However, in the following counter abstraction this formulization has drawbacks, due to the non-determinism of the operations on the abstract domain, while the abstraction of 0 is precise.

Counter abstraction. The counter representation encodes a system of $n - f$ processes as a single process system. When, n, t, and f are fixed, the elements of array k are bounded by n. However, in the parameterized case the elements of k are unbounded. To circumvent this problem, we apply the PIA abstraction from Section 4.1 to the elements of k.

In the counter abstraction, the elements of k range over the abstract domain $\widehat{D}$. Similar to Section 4.1, we have to compute the abstract operations over k. These are the operations in the *init* block and in the *update* block.

To transform the *init* block, we first compute the existential abstraction of $\sum_{\ell \in L_0} k[\ell] = N(\mathbf{p})$. In our example, we compute the set $||k[0] + k[3] = n - f||_\exists$ and non-deterministically choose an element from this set. Again, we can do it with Yices. We give the initialization block after the abstraction (note that the number of choices is fixed and determined by the size of the abstract domain):

```
if /* 0 ↦ (pc = V0, nrcvd = I_0); 3 ↦ (pc = V1, nrcvd = I_0) */
  :: k[0] = 3 /* I_{n−t} */; k[3] = 0 /* I_0 */;
  :: k[0] = 3 /* I_{n−t} */; k[3] = 1 /* I_1 */;
  ...
  :: k[0] = 0 /* I_0 */;   k[3] = 3 /* I_{n−t} */;
fi;
```

Listing 12: Initialization of the counters

In the *update* block we have to compute the abstraction of `k[4 * pc + nrcvd]--` and `k[4 * next_pc + next_nrcvd]++`. We have already seen how to do this with the data abstraction. The update block looks as follows after the abstraction:

```
if
  :: pc != next_pc || nrcvd != next_nrcvd ->
    if /* decrement the counter of the previous state */
      :: (k[((pc * 4) + nrcvd)] == 3) ->
        k[((pc * 4) + nrcvd)] = 3;
      :: (k[((pc * 4) + nrcvd)] == 3) ->
        k[((pc * 4) + nrcvd)] = 2;
      ...
      :: (k[((pc * 4) + nrcvd)] == 1) ->
        k[((pc * 4) + nrcvd)] = 0;
    fi;
    if /* increment the counter of the next state */
      :: (k[((next_pc * 4) + next_nrcvd)] == 3) ->
        k[((next_pc * 4) + next_nrcvd)] = 3;
      :: (k[((next_pc * 4) + next_nrcvd)] == 2) ->
        k[((next_pc * 4) + next_nrcvd)] = 3;
      ...
      :: (k[((next_pc * 4) + next_nrcvd)] == 0) ->
        k[((next_pc * 4) + next_nrcvd)] = 1;
    fi;
  :: else; /* do not update the counters */
fi;
```

Listing 13: Abstract increment and decrement of the counters

In contrast to the counter representation, the increment and decrement of the counters in the array k are now non-deterministic. For instance, the counter `k[((pc * 4) + nrcvd)]` can change its value from I_{n-t} to I_{t+1} or stay unchanged. Similarly, the value of `k[((next_pc * 4) + next_nrcvd)]` can change from I_1 to I_{t+1} or stay unchanged.

Observe that this non-determinism adds behaviors to the abstract systems:

- both counters could stay unchanged, which leads to stuttering
- the value of `k[((pc * 4) + nrcvd)]` decreases, while at the same time the value of `k[((next_pc * 4) + next_nrcvd)]` stays unchanged, that is, we lose processes, and finally
- `k[((pc * 4) + nrcvd)]` stays unchanged and `k[((next_pc * 4) + next_nrcvd)]` increases, that is, processes are added.

Some of these behaviors lead to spurious counterexamples we deal with in Section 5. Figure 8 gives a small part of the transition system obtained from the counter abstraction. We omit local states that have the counter value I_0 to facilitate reading. 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 $sv = \texttt{V1}$. 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.

Specifications. Similar to the counter representations, quantifiers can be encoded as expressions on the counters. Instead of comparing to 0, we compare to the abstract zero I_0: A global state satisfies $[\exists i.\ \Phi(i)]$ if $\bigvee_{\ell \in [\![\Phi]\!]} k[\ell] \neq I_0$. Similarly, a global state satisfies $[\forall i.\ \Phi(i)]$ if $\bigwedge_{\ell \notin [\![\Phi]\!]} k[\ell] = I_0$.

4.3 Soundness

We do not focus on the soundness proofs here, the details can be found in [52]. The soundness is based on two properties:

First, between every concrete system and the abstract system, there is a simulation relation. The central argument to prove this comes from Proposition 2, from which follows that if a threshold is satisfied in the concrete system, the abstraction of the threshold is satisfied in the abstract systems. Intuitively, this means that if a transition is enabled in the concrete system, then it is enabled in the abstract system, which is required to prove simulation.

Second, the abstraction of a fair path (with respect to our justice properties) in the concrete system is a fair path in the abstract system. This follows from construction: we label an abstract state with a proposition if the abstract state satisfies the existential abstraction of the proposition, in other words, if there is a concretization of the abstract state that satisfies the proposition.

5 Abstraction refinement

In Sections 4.1 and 4.2, we constructed approximations of the transition systems: First, we transformed parameterized code of a process into finite-state non-parameterized code; Second, we constructed a finite-state process that approximates the behavior of $n - f$ processes. Usually, abstraction introduces new behavior that is not present in the concrete system. As a result, specifications that hold in the concrete system, may be violated in the abstract system. In this case, a model checker returns an execution of the abstract system that cannot be replayed in the concrete system; such an execution is called a *spurious counterexample*.

As it was suggested in Proposition 2, PIA data abstraction does not lose precision for the comparisons against threshold expressions. In fact, in our experiments we have not seen spurious counterexamples caused by the PIA data abstraction. So, we focus on abstraction refinement of the PIA counter abstraction, where we have identified three sources of spurious behavior (a) the run contains a transition where the number of processes is decreasing or increasing; (b) the number of messages sent by processes deviates from the number of processes who have sent a message; (c) unfair loops.

Given a run of the counter abstraction, we have to check that the run is spurious for all combinations of parameters from $\mathbf{P}_{RC}$. This problem is again parameterized, and we are not aware of techniques to deal with it in the general case. Thus, we limit ourselves to detecting the runs that have a *uniformly spurious transition*, that is, a transition that does not have a concretization for all the parameters from $\mathbf{P}_{RC}$.

We check for spurious transitions using SMT solvers. To do so, we have to encode the transition relation of all concrete systems (which are defined by different parameter values) in SMT. We explain our approach in three steps: first we encode a single PROMELA statement. Based on this we encode a process step that consist of several statements. Finally, we use the encoding of a step to define the transition relation of the system.

5.1 Encoding the transition relation

Encoding a single statement. As we want to detect spurious behavior, the SMT encoding must capture a system on a less abstract level than the counter abstraction. One first idea would be to encode the transition relation of the concrete systems. However, as we do parameterized model checking, we actually have infinitely many concrete systems, and the state space and the number of processes in these systems is not bounded. Hence, we require a representation whose "degree of abstraction" lies between the concrete systems and the counter abstraction. In principle, the counter representation from Section 4.2 seems to be a good candidate. Its state is given by finitely many integer counters, and finitely many shared variables that range over the abstract domain. Although there are infinitely many states (the counters are not bounded), the state space and transition relation can be encoded as an SMT problem. Moreover, threshold guards and the operations on the process counters can be expressed in linear integer arithmetic, which is supported by many SMT solvers.

However, experiments showed that we need a representation closer to the concrete systems. Hence, we use a system whose only difference to the counter representation from Section 4.2 is that the shared variables are not abstracted. The main difficulty in this is to encode transitions that involve abstract local as well as concrete global variables. For that, we represent the parameters in SMT. Then, instead of comparing global variables against abstract values, we check whether the global variables are within parametric intervals. Here we do not go into the formal details of this abstraction. Rather, we explain it by example. The most complicated case is the one where an expression involves the parameters, local variables, and shared variables. For instance, consider the code on page 25, where a is a local variable and b is a shared one. In this new abstraction a is abstract and b is concrete. Thus, we have to encode constraints on b as inequalities expressing which interval b belongs to. Specifically, we replace $b = I_k$ with either $e_k \leq b < e_{k+1}$ (when k is not the largest threshold e_μ), or $e_\mu \leq b$ (otherwise):

```
    $a == I_{n-t}$ && $n-t \le b$ || $a == I_{t+1}$ && $n-t \le b$
 || $a == I_1$ && $n-t \le b$ || $a == I_0$ && $n-t \le b$
 || $a == I_{n-t}$ && $t+1 \le b < n-t$ || $a == I_{t+1}$ && $t+1 \le b < n-t$
 || $a == I_1$ && $t+1 \le b < n-t$ || $a == I_0$ && $t+1 \le b < n-t$
 || $a == I_{t+1}$ && $1 \le b < t+1$ || $a == I_1$ && $1 \le b < t+1$
 || $a == I_0$ && $1 \le b < t+1$ || $a == I_1$ && $0 \le b < 1$
 || $a == I_0$ && $0 \le b < 1$
```

Apart from this, statements that depend solely on shared variables are not changed. Finally, statements that consist of local variables and parameters are abstracted as in Section 4.1. This level of abstraction allows us to detect spurious transitions of both types (a) and (b).

Encoding a single process step. Our Promela code defines a transition system: A single iteration of the loop expresses one step (or transition) which consists of several expressions executed indivisibly. The code before the loop defines the constraints on the initial states of the transition system. Recall that we can express Promela code as a control flow automaton (cf. Section 3.3 and Figure 4). Formally, a *guarded control flow automaton* (CFA) is an edge-labeled directed acyclic graph $A = (Q, q_I, q_F, E)$ with a finite set Q of nodes called locations, an initial location $q_I \in Q$, and a final location $q_F \in Q$. Edges are labeled with simple Promela statements (assignments and comparisons). Each transition is defined by a path from q_I to q_F in a CFA. Our goal is to construct a formula that encodes the transition relation. We are doing this by translating a statement on every edge from E into an SMT formula in a way similar to [17][Ch. 16]. What we show below is not the most efficient encoding, but we omit optimizations to keep presentation clear.

First, we have to take care of multiple assignments to the same variable, as they can overwrite previously assigned values. Consider the following sequence of Promela statements `S: x=1; y=x; x=2; z=x` corresponding to a path of some CFA. If we naively encode it as $x = 1 \wedge y = x \wedge x = 2 \wedge z = x$, then the formula is immediately unsatisfiable due to the conflicting constraints on x. We thus need multiple versions of such variables, that is, we turn sequence `S` into `S1: x1=1; y1=x1; x2=2; z1=x2`. Now we can construct formula $x^1 = 1 \wedge y^1 = x^1 \wedge x^2 = 2; z^1 = x^2$ that treats the assignments correctly. Such a form is known as static single assignment (SSA); it can be computed by an algorithm given in [34].

We assume the following notation for the multiple copies of a variable x in SSA: x denotes the *input* variable, that is, the copy of x at location q_I; x' denotes the *output* variable, that is, the copy of x at location q_F; x^i denotes a *temporary* variable, that is, a copy of x that is overwritten by another copy before reaching q_F.

From now on we assume that the CFA in given in SSA form, and we can thus encode the transition relation. This requires us to capture all paths of the CFA. Our goal is to construct a single formula T over the following vectors of free variables:

- $\mathbf{p}$ is the vector of integer parameters from Π, which is not changed by a transition;
- $\boldsymbol{x}$ is the vector of integer input variables from $\Lambda \cup \{sv\}$;
- $\boldsymbol{x'}$ is the vector of integer output variables of $\boldsymbol{x}$;
- $\boldsymbol{g}$ is the vector of integer input variables from Γ;
- $\boldsymbol{g'}$ is the vector of integer output variables of $\boldsymbol{g}$;
- $\boldsymbol{t}$ is the vector of integer temporary variables of $\boldsymbol{x}$ and $\boldsymbol{g}$;
- $\boldsymbol{en}$ is the vector of boolean variables, one variable en_e per an edge $e \in E$, which means that edge e lies on the path from q_I to q_F.

Let $\mathit{form}(s)$ be a straightforward translation of a Promela statement s into a formula as discussed above. Assignments are replaced with equalities and relations (e.g., $\leq$, $>$) are kept as they are. Then, for an edge $e \in E$ labeled with a statement s, we construct a formula $T_e(\mathbf{p}, \boldsymbol{x}, \boldsymbol{x'}, \boldsymbol{g}, \boldsymbol{g'}, \boldsymbol{t}, \boldsymbol{en})$ as follows:

$$T_e \equiv en_e \rightarrow \mathit{form}(s),$$

Now, formula T is constructed as the following conjunction whose subformulas are discussed in detail below:

$$T \equiv \mathit{start} \wedge \mathit{follow} \wedge \mathit{mux} \wedge \bigwedge_{e \in E} T_e$$

Intuitively, *start* is saying that at least one edge outgoing from q_I is activated; *follow* is saying that whenever a location has an incoming activated edge, it also has at least one outgoing activated edge; *mux* is expressing the fact that at most one outgoing edge can be picked. Formally, the formulas are defined as follows:

$$\begin{aligned}
\mathit{start} &\equiv \bigvee_{(q,q') \in E:\ q = q_I} en_{(q,q')} \\
\mathit{follow} &\equiv \bigwedge_{(q,q') \in E} \left(en_{(q,q')} \rightarrow \bigvee_{(q',q'') \in E} en_{(q',q'')} \right) \\
\mathit{mux} &\equiv \bigvee_{(q,q'),(q,q'') \in E} \neg en_{(q,q')} \vee \neg en_{(q,q'')}
\end{aligned}$$

We have to introduce formula *mux*, because the branching operators in Promela allow one to pick a branch non-deterministically, whenever the guard of the branch evaluates to true. To pick exactly one branch, we have to introduce the mutual exclusion constraints in the form of *mux*. In contrast, programming languages like C do not need this constraint, as the conditions of the if-branch and the else-branch cannot both evaluate to true simultaneously.

Having constructed formula T, we say that a process can make a transition from state $\boldsymbol{x}$ to state $\boldsymbol{x'}$ under some combination of parameters if and only if the formula $T(\mathbf{p}, \boldsymbol{x}, \boldsymbol{x'}, \boldsymbol{g}, \boldsymbol{g'}, \boldsymbol{t}, \boldsymbol{en}) \wedge RC(\mathbf{p})$ is satisfiable.

Transition relation of the counter representation. Now we show how to encode the transition relation R of the counter representation using the process transition relation T. The transition relation connects counters $\boldsymbol{k}$ and global variables $\boldsymbol{g}$ before a step with their primed versions $\boldsymbol{k}'$ and $\boldsymbol{g}'$ after the step. Recall that in Section 4.2, we introduced bijection h that maps states to numbers. In the following, by abuse of notation, by $h(\boldsymbol{x})$ we denote an SMT expression that encodes the bijection h. We will use the formulas *dec* and *inc*: Informally, *dec* ensures that the counter that corresponds to $h(\boldsymbol{x})$ is not equal to zero and decrements the counter, while *inc* increments the counter $\boldsymbol{k}[h(\boldsymbol{x}')]$. Formula R is the following conjunction

$$R \equiv \mathit{dec} \wedge T \wedge \mathit{inc} \wedge \mathit{keep},$$

and we define *dec*, *inc*, and *keep* as follows:

$$\begin{aligned}
\mathit{dec} &\equiv \bigwedge_{0 \le \ell < |L|} h(\boldsymbol{x}) = \ell \rightarrow \boldsymbol{k}[\ell] > 0 \wedge \boldsymbol{k}'[\ell] = \boldsymbol{k}[\ell] - 1 \\
\mathit{inc} &\equiv \bigwedge_{0 \le \ell < |L|} h(\boldsymbol{x}') = \ell \rightarrow \boldsymbol{k}'[\ell] = \boldsymbol{k}[\ell] + 1 \\
\mathit{keep} &\equiv \bigwedge_{0 \le \ell < |L|} (h(\boldsymbol{x}) \ne \ell \wedge h(\boldsymbol{x}') \ne \ell) \rightarrow \boldsymbol{k}'[\ell] = \boldsymbol{k}[\ell]
\end{aligned}$$

Now we can say that a counter representation of a system makes a step from $(\boldsymbol{k}, \boldsymbol{g})$ to $(\boldsymbol{k}', \boldsymbol{g}')$ if and only if $R(\mathbf{p}, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{k}, \boldsymbol{k}', \boldsymbol{g}, \boldsymbol{g}', \boldsymbol{t}, \boldsymbol{en}) \wedge RC(\mathbf{p})$ is satisfiable. In what follows, we denote the latter formula by *Step*.

In order to encode operations on $\boldsymbol{k}$, we are using arrays. In our case, however, each array may be replaced with $|L|$ integer variables. Thus, we do not actually use important properties of array theory.

5.2 Spurious Behavior

Losing and introducing processes. We start with the first type of spurious behavior, where a transition "loses" or "introduces" processes. Consider the following sequence of abstract states, which introduces new processes due to nondeterminism of the counter updates:

```
1 k = {0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
2 k = {0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}, nsnt =1
3 k = {0, 0, 0, 0, 3, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0}, nsnt =2
```

Here we represent the abstract states in the format similar to the one used in our tool (Section 6). The assignment "$k = \{\dots\}$" shows the contents of the array k in C format, that is, the position $i = h(\ell)$ gives the abstract number of processes in local state ℓ. The assignment "$nsnt = \dots$" shows the value of $nsnt$.

As one can see, counter `k[8]` changes its value from I_0 to I_1 and then to I_{t+1}. The combination of `k[8]` $= I_{t+1}$ and `k[4]` $= I_{n-t}$ indicates that the transition from state 2 to state 3 is spurious. In fact, we can detect this kind of spurious behavior with Yices:

```
1  (set-evidence! true)
2  (set-verbosity! 3)
3  (define n::int)
4  (define t::int)
5  (define f::int)
6  (assert (and (> n (* 3 t)) (> t f) (>= f 0)))
7  (define k :: (-> (subrange 0 15) nat))
8  (assert+ (and (<= (- n t) (k 4))))
9  (assert+ (and (<= (+ t 1) (k 8)) (< (k 8) (- n t))))
10 ;; -> copy the assertion below for the indices 1-3, 5-7, 9-15
11 (assert+ (and (<= 0 (k 0)) (< (k 0) 1)))
12 (assert (= (- n f) (+
13   (k 0) (k 1) (k 2) (k 3) (k 4) (k 5) (k 6) (k 7)
14   (k 8) (k 9) (k 10) (k 11) (k 12) (k 13) (k 14) (k 15))))
15 (check)
```

Listing 14: Constraints on state 3 encoded in Yices

In lines (8) – (11), we constrain the values of process counters to reside within the parametric intervals as defined by the abstract values of state 3. In lines (12) – (14), we assert that the total number of processes equals $n - f$. Yices reports that the constraints are unsatisfiable, which means that state 3 cannot be an abstraction of a system state with $n-f$ processes. We conclude that the transition from state 2 to state 3 is uniformly spurious, and we eliminate it.

In fact, Yices also reports that it did not use all the assertions to come up with unsatisfiability. An unsatisfiable core — a minimal set of assertions that leads to unsatisfiability — consists of the assertions in lines (8) – (9). Thus, we can remove every transition leading to a state with $k[4] = I_{n-t}$ and $k[8] = I_{t+1}$.

Now consider a sequence of abstract states, which is losing processes:

```
1 k = {0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
2 k = {0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}, nsnt =1
3 k = {0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}, nsnt =1
```

As with the case of introducing processes, we can detect with Yices that the transition from state 2 to state 3 is uniformly spurious, and eliminate all the transitions captured with an unsatisfiable core.

Losing messages. In our case study (cf. Figure 4) 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 is in $\{\mathrm{SE}, \mathrm{AC}\}$, while due to phenomena similar to those discussed above, we can "lose messages" in the abstract system. When checking safety properties, this kind of spurious behavior does not produce counterexamples. However, it generates spurious counterexamples for liveness. Consider the following example:

```
1 k = {0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
2 k = {0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}, nsnt =1
3 k = {0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0}, nsnt =1
```

Consider state 3. Here, the number of processes with sv = SE is at least $t+1$ (as $k[8] = 2$ corresponding to I_{t+1}), while the number of messages is always strictly less than $t + 1$ (as $nsnt = 1$ corresponding to I_1). We can try to check, whether the transition from state 2 to state 3 is spurious. This time, we also add the constraints by *Step*:

```
1  (set-evidence! true)
2  (set-verbosity! 3)
3  (define n::int)
4  (define t::int)
5  (define f::int)
6  (assert (and (> n (* 3 t)) (> t f) (>= f 0)))
7  (define k :: (-> (subrange 0 15) nat))
8  (define k' :: (-> (subrange 0 15) nat))
9  (assert+ (and (<= (+ t 1) (k 4)) (<= (k 4) (- n t) )))
10 (assert+ (and (<= 1 (k 8)) (< (k 8) (+ t 1))))
11 (assert+ (and (<= 1 (k' 4)) (<= (k' 4) (+ t 1) )))
12 (assert+ (and (<= 1 (k' 8)) (< (k' 8) (+ t 1))))
13 ;; copy the assertions below for the indices 1-3, 5-7, 9-15
14 (assert+ (and (<= 0 (k 0)) (< (k 0) 1)))
15 (assert+ (and (<= 0 (k' 0)) (< (k' 0) 1)))
16 ;; -> copy Step here <-
17 (check)
```

Listing 15: Concretization of transition from state 2 to state 3 in Yices

This time Yices reports that the constraints are satisfiable. Indeed, it is possible to pick the number of processes that satisfy the constraints in lines (9)–(12) in Listing 15 and still do not increase `nsnt` so that it reaches $t + 1$. As we know that this example represents spurious behavior, the user can introduce an invariant candidate in Promela:

```
atomic tx_inv =
    ((card(Proc:pc == SE) + card(Proc:pc == AC)) == nsnt);
```

Then we can automatically test, whether the invariant candidate *tx_inv* is an invariant by checking that the following formula is unsatisfiable (tx_inv' is a copy of *tx_inv* with $\boldsymbol{x}$ replaced by $\boldsymbol{x}'$, and *Init* is a formula encoding the initial states):

$$\neg((Init \rightarrow tx\_inv) \wedge ((tx\_inv \wedge Step) \rightarrow tx\_inv'))$$

As soon as we know that `tx_inv` is an invariant, we can add the following assertion to the previous query in Yices:

```
18 (assert (= nsnt (+
19   (k 8) (k 9) (k 10) (k 11) (k 12) (k 13) (k 14) (k 15))))
```

Listing 16: Constraint expressed by the invariant *tx_inv*

With this assertion in place, we discover that the transition from state 2 to states 3 is uniformly spurious.

5.3 Removing transitions in Promela

So far, we have been concerned with detecting uniformly spurious transitions. Now we discuss how one can remove spurious transitions from the counter abstraction that we introduced in Section 4.2 (cf. code on p. 27).

Whenever we detect a uniformly spurious transition, we extract two sets of constraints from the SMT solver: The constraints on the abstract state before the transition (precondition), and the constraints on the abstract state after the transition (postcondition). Consider the following uniformly spurious transition:

```
k = {1, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =0
k = {1, 1, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt =1
```

Here we extract the following constraints from an unsatisfiability core given to us by the SMT solver (written in PROMELA notation):

```
pre = (nsnt == 0);
post = (k[0] == 1) && (k[1] == 2)
  && (k[4] == 3) && (k[15] == 0) && (nsnt == 0);
```

In order to remove the spurious transition, we have to enforce SPIN to prune the executions that include the transition. To this end, we introduce a boolean variable `is_spur` that turns true, whenever the current execution has at least one spurious transition. Then for each refinement iteration $K \geq 1$ we introduce a boolean variable `pK_pre` that turns true, whenever the current state satisfies the precondition of the spurious transition detected in iteration K. We modify PROMELA code as follows:

```
bool is_spur = 0; /* is the current execution spurious */
bool p1_pre = 0;  /* detected at refinement iteration 1 */
...
bool pK_pre = 0;  /* detected at refinement iteration K */
...
active[1] proctype CtrAbs() {
  ...
  /* init */
loop:
  ...
  pK_pre = (nsnt == 0);
  /* select */
  /* receive */
  /* compute */
  /* send */
  /* update counters */
  ...
  /* is the current transition spurious? */
  spur = spur || pK_pre && k[0] == 1 && k[1] == 2
       && k[4] == 3 && k[15] == 0 && nsnt == 0;
  goto loop;
}
```

Listing 17: Counter abstraction with detection of spurious transitions

Finally, we prune the spurious executions by modifying each LTL\X formula φ in Promela specifications as follows:

```
[]!is_spur -> φ
```

5.4 Detecting unfair loops

There is a third kind of spurious behavior that is not present in our case study, but it occurs in the experiments with omission faults (cf. Section 6). Modeling omission faults introduces 12 local states instead of 16. Here is a counterexample showing the violation of liveness property R (cf. Section 3):

```
 3  k = {2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0}, nsnt = 0
 4  k = {2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0}, nsnt = 1
 5  k = {2, 0, 0, 2, 0, 0, 1, 1, 0, 0, 0, 0}, nsnt = 2
 6  k = {2, 0, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0}, nsnt = 2
 7  k = {1, 0, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0}, nsnt = 2
 8  k = {0, 0, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0}, nsnt = 2
 9  k = {0, 0, 0, 1, 0, 0, 1, 2, 0, 0, 0, 0}, nsnt = 2
10  k = {0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0}, nsnt = 2
11  k = {0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0}, nsnt = 2
12  k = {0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1}, nsnt = 2
13  k = {0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2}, nsnt = 2
14  k = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2}, nsnt = 2
15  <<<<<START OF CYCLE>>>>>
16  k = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2}, nsnt = 2
```

Listing 18: A counterexample with a spurious (unfair) loop

Here state (16) is repeated in a loop, but it violates the following fairness constraint saying that up to *nsnt* $-$ *F* messages must be eventually delivered:

```
atomic in_transit = some(Proc:nrcvd < nsnt - F);
ltl fairness { []<>(!in_transit) && (...) }
```

Again, using the SMT solver we can check, whether the loop is unfair, that is, no state within the loop satisfies the fairness constraint, e.g., !in_transit.

```
(set-evidence! true)
(set-verbosity! 3)
(define n::int)
(define t::int)
(define f::int)
(assert (and (> n (* 2 t)) (> t f) (>= f 0)))
(define k :: (-> (subrange 0 11) nat))
;; the constraints by the state 14:
(assert+ (and (<= 1 (k 7)) (< (k 4) (+ t 1) )))
(assert+ (and (<= (+ t 1) (k 11))))
;; -> repeat the assertion below for the indices 0-6, 7-10
(assert+ (and (<= 0 (k 0)) (< (k 0) 1)))
(assert+ (>= nsnt (+ t 1)))
```

```
;; constraints by !in_transit
(assert+ (not (or
 (and (>= (- nsnt f) (+ t 1))
   (or (/= (k 1) 0) (/= (k 4) 0) (/= (k 7) 0) (/= (k 10) 0)))
 (and (>= (- nsnt f) (+ t 1))
   (or (/= (k 0) 0) (/= (k 3) 0) (/= (k 6) 0) (/= (k 9) 0)))
 (and (>= (- nsnt f) 1) (< (- nsnt f) (+ t 1))
    (or (/= (k 0) 0) (/= (k 3) 0) (/= (k 6) 0) (/= (k 9) 0)))
)))
(check)
```

Listing 19: Does state 16 have a concretization that meets justice constraints?

This query is unsatisfiable and Yices gives us an unsatisfiable core that we track in Promela as we did with the spurious transitions:

```
/* update counters */
...
r0 = k[0] == 0 && k[1] == 0
  && k[2] == 0 && k[3] == 0 && k[4] == 0
  && k[5] == 0 && k[7] == 1 && k[10] == 0;
```

and modify each specification φ to avoid infinite occurrences of `r0`:

```
  (<>[]r0) || φ
```

6 Experiments

In this section we describe the tool chain ByMC implementing the approach presented in Sections 3 – 5. We also demonstrate the results of experiments on finite-state as well as parameterized model checking of fault-tolerant distributed algorithms. The tool and the benchmarks are available at [1].

6.1 Running the Tool

In what follows, we use the tool on the running example `bcast-byz.pml` available in the set of benchmarks `benchmarks-sfm14` at [1]. We also assume that the tool resides in the directory `${bymc}`.

The tool chain supports two modes of operation:

- **Concrete model checking.** In this mode, the user fixes the values of the parameters $\mathbf{p}$. The tool instantiates code in standard Promela and performs finite-state model checking with Spin. This step is very useful to make sure that the user code operates as expected without abstraction involved.
- **Parameterized model checking**. In this mode, the tool applies data and counter abstractions (cf. Section 4), and performs finite-state model checking of the abstract model with Spin.

For concrete-state model checking of the `relay` property, one issues command `verifyco-spin` as follows:

```
$ ${bymc}/verifyco-spin "N=4,T=1,F=1" bcast-byz.pml relay
```

The tool instantiates the model checking problem in directory "`./x/spin-bcast-byz-relay-N=4,T=1,F=1`". The directory contains file `concrete.prm` that differs from the source code as follows: The parameters N, T, and F in the Promela code are replaced with the values 4, 1, 1 respectively. The process prototype is replaced with $N - F = 3$ active processes.

In order to run parameterized model checking, one issues `verifypa-spin` as follows:

```
$ ${bymc}/verifypa-spin bcast-omit.pml relay
```

The tool instantiates the model checking problem in a directory, whose name follows the pattern "`./x/bcast-byz-relay-yymmdd-HHMM.*`". The directory contains the following files of interest: `abs-interval.prm` is the result of the data abstraction; `abs-counter.prm` is the result of the counter abstraction; `abs-vass.prm` is the auxiliary abstraction for the abstraction refinement; `mc.out` contains the last output by Spin; `cex.trace` contains the counterexample (if there is one); `yices.log` contains communication log with Yices.

6.2 Concrete Model Checking for Small System Sizes

Listing 3 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 [11]) 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 [23]. Further, we verified a Byzantine tolerant broadcasting algorithm from [20]. For the encoding of the algorithm from [20] we were required to use two message types and thus two shared variables — opposed to the one type of the ⟨echo⟩ messages in Algorithm 1. Finally, we implemented the asynchronous condition-based consensus algorithm from [67]. We specialized it to binary consensus, which resulted in an encoding which requires four shared variables.

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 [76], for each of which we systematically changed the parameter values. By doing so, we verify that under our modeling the different combination of parameters lead

#	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

Table 1: Summary of experiments related to [76]

to the expected result. Table 1 and Figure 9 summarize the results of our experiments for broadcasting algorithms in the spirit of [76]. 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 [69], 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 [23], [20], and [67]. The specification (F) is related to agreement and was also used in [43]. 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

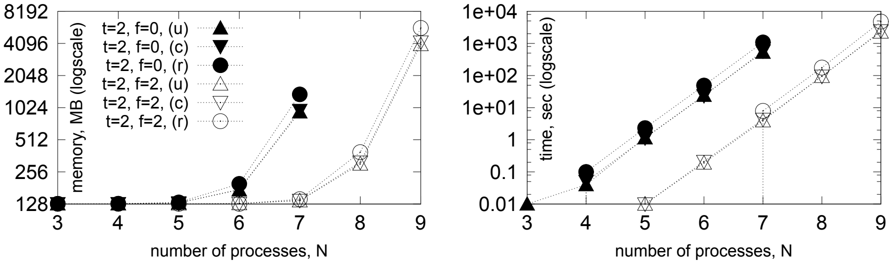


Fig. 9: SPIN memory usage (left) and running time (right) for BYZ.

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.

6.3 Parameterized Model Checking

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 4, for t Byzantine faults if $n > 3t$, (SYMM) for t symmetric (identical Byzantine [11]) faults if $n > 2t$, (OMIT) for t send omission faults if $n > 2t$, and (CLEAN) for t clean crash faults [80] if $n > t$. In addition, we verified the folklore broadcasting algorithm FBC — formalized also in [43] — whose CFA is given in Figure 5.

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

Table 3 summarizes our experiments run on 3.3GHz Intel® Core$^{\text{TM}}$ 4GB. In the cases (A) we used resilience conditions as provided by the literature, and verified the specification. The model FBC is the folklore reliable broadcast algorithm also considered in [43] under the resilience condition $n \geq t \geq f$. In the bottom part of Table 3 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

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
Folklore Broadcast [23]								
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 [20]								
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 [67]								
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

Table 2: Summary of experiments with algorithms from [23,20,67]

Spin running time after the last refinement step. Finally, column $|\widehat{D}|$ indicates the size of the abstract domain.

7 Discussions

Input languages for software model checkers are designed to capture limited degrees of non-determinism that are required to check, e.g., C/C++ industrial software. However, distributed algorithms typically show higher degrees of non-determinism, which makes them challenging for such existing tools. Promela, the input language of the Spin model checker [2], was designed to simulate and validate network protocols. Consequently, Promela contains several primitives for concurrent and distributed systems, and we consider it the most suitable language for our purposes. Still, as we discussed, the semantics of the constructs do not match the ones required by distributed algorithms, and straight-forward implementations do not scale well. Similarly, PlusCal [60] is a high-level language to describe algorithms that can be translated to TLA+. It contains constructs to specify concurrent systems with shared variables. The UPPAAL model checker [14] has channels that model synchronous communications similar to rendezvous. Besides, it contains a broadcast primitive that is more closely related to hardware than to broadcasts in distributed systems. The input for the SMV model checker is also oriented towards hardware and provides rather low-level

$M \models \varphi$?	RC	**Spin**	**Spin**	**Spin**	**Spin**	$\lvert\widehat{D}\rvert$	**#R**	**Total**
		Time	**Memory**	**States**	**Depth**			**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
Symm $\models U$	(a)	0.1 s	67 MB	19k	897	3	0	1 s
Symm $\models C$	(a)	0.1 s	67 MB	19k	1113	3	2	3 s
Symm $\models R$	(a)	0.3 s	69 MB	87k	2047	3	12	16 s
Omit $\models U$	(a)	0.1 s	66 MB	4k	487	3	0	1 s
Omit $\models C$	(a)	0.1 s	66 MB	7k	747	3	5	6 s
Omit $\models R$	(a)	0.1 s	66 MB	8k	704	3	5	10 s
Clean $\models U$	(a)	0.3 s	67 MB	30k	1371	3	0	2 s
Clean $\models C$	(a)	0.4 s	67 MB	35k	1707	3	4	8 s
Clean $\models R$	(a)	1.1 s	67 MB	51k	2162	3	13	31 s
FBC $\models U$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
FBC $\models F$	—	0.1 s	66 MB	1.7k	333	2	0	1 s
FBC $\models R$	—	0.1 s	66 MB	1.2k	259	2	0	1 s
FBC $\not\models C$	—	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
Symm $\not\models U$	(b)	0.1 s	67 MB	19k	947	3	0	2 s
Symm $\not\models C$	(b)	0.1 s	67 MB	18k	1175	3	2	4 s
Symm $\models R$	(b)	0.2 s	67 MB	42k	1681	3	8	12 s
Omit $\models U$	(d)	0.1 s	66 MB	5k	487	3	0	1 s
Omit $\not\models C$	(d)	0.1 s	66 MB	5k	487	3	0	2 s
Omit $\not\models R$	(d)	0.1 s	66 MB	0.1k	401	3	0	2 s

Table 3: Summary of experiments in the parameterized case.

communication and coordination primitives. Lustre [48] is the input language for the SCADE tool set, and is limited to tightly coupled synchronous systems.

There have been two major undertakings of formalization that gained acceptance within the distributed algorithms community. Both were initiated by researchers with a background in distributed algorithms and with a precise understanding of what needs to be expressed. These approaches are on the one hand, the I/O Automata by Lynch and several collaborators [63,55,66], and on the other hand, TLA by Lamport and others [59,54,60]. IOA and TLA are general frameworks that are based on labeled transition systems and a variant of linear temporal logic, respectively. Both frameworks were originally developed at a time when automated verification was out of reach, and they were mostly intended to be used as formal foundations for handwritten proofs. Today, the tool support for IOA is still in preliminary stages [3]. For TLA [4], the TLC model checker is a simple explicit state model checker, while the current version of the TLA+ Proof System can only check safety proofs.

In all these approaches, specifying the semantics for fault-tolerant distributed algorithms is a research challenge, and we believe that this research requires an interdisciplinary effort between researchers in distributed algorithms and model checking. In this tutorial we presented our first results towards this direction.

The automatic verification of state-of-the-art distributed algorithm such as Paxos [58], or even more importantly, their implementations are currently out of reach, except possibly for very small system sizes. To be eventually able to verify such algorithms, we have to find efficient means to address the many problems these distributed algorithms pose to verification, for instance, large degrees of non-determinism due to faults and asynchrony, parameterization, and the use of communication primitives that are non-standard to the verification literature. The work that is presented here provides first steps in this direction. We focused on a specific class of fault-tolerant distributed algorithms, namely, threshold-based algorithms and derived abstraction methods for them.

The only way to evaluate the practical use of an abstraction is to conduct experiments on several case studies, and thus demonstrate that the abstraction is sufficiently precise to verify correct distributed algorithms, and find counterexamples in buggy ones. Hence, understanding implementations is crucial to evaluate the theoretical work and they are thus of highest importance. This motivates this tutorial that discussed the abstraction methods from an implementation point of view.

In more detail, we first added mild additions to the syntax of Promela to be able to express the kind of parameterized systems we are interested in. We also showed by experimental evaluation that the standard language constructs for interprocess communication do not scale well, and do not naturally match the required semantics for fault-tolerant distributed algorithms. We thus introduced an efficient encoding of a fault-tolerant distributed algorithm in the extended Promela. This representation builds the input for our tool chain, and we discussed in detail how it can be automatically translated into abstract models. We have introduced several levels of abstractions. As our abstractions are over-approximations, the model checker returned spurious counterexamples, such that we were led to counter example guided abstraction refinement (CEGAR) [27]. In contrast to the classic CEGAR setting, in the parameterized case we have an infinite number of concrete systems which poses new challenges. In this paper we discussed several of them and presented the details of the abstraction refinement approach that was sufficient to verify some of our case studies.

When taking a close look at our experiments, one observes that there are several algorithms that we verified for small instances, while we could not verify them in the parameterized setting. Developing new methods that allow us to also verify them is subject to ongoing work.

Acknowledgments. We are grateful to Francesco Spegni whose constructive comments helped us to improve the presentation.

References

1. ByMC 0.4.0: Byzantine model checker (2013), `http://forsyte.tuwien.ac.at/software/bymc/`, accessed: March, 2014
2. Spin 6.2.7 (2014), `http://spinroot.com/`, accessed: March, 2014
3. Tempo toolset. Web page, `http://www.veromodo.com/`
4. TLA – the temporal logic of actions. Web page, `http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html`
5. Yices 1.0.40 (2013), `http://yices.csl.sri.com/yices1-documentation.shtml`, accessed: March, 2014
6. Abdulla, P.A.: Regular model checking. International Journal on Software Tools for Technology Transfer 14, 109–118 (2012)
7. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. 127(2), 91–101 (1996)
8. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with Byzantine failures and little system synchrony. In: DSN. pp. 147–155 (2006)
9. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. JSAT 8(1/2), 29–61 (2012)
10. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 15, 307–309 (1986)
11. Attiya, H., Welch, J.: Distributed Computing. John Wiley & Sons, 2nd edn. (2004)
12. Baier, C., Katoen, J.P., Larsen, K.G.: Principles of Model Checking. MIT Press (2008)
13. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI. pp. 203–213 (2001)
14. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal 4.0 (2006)
15. Biely, M., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper, A., Widder, J.: Tolerating corrupted communication. In: PODC. pp. 244–253 (Aug 2007)
16. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theoretical Computer Science 412(40), 5602–5630 (2011)
17. Biere, A.: Handbook of satisfiability, vol. 185. IOS Press (2009)
18. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: DAC. pp. 317–320 (1999)
19. Bokor, P., Kinder, J., Serafini, M., Suri, N.: Efficient model checking of fault-tolerant distributed protocols. In: DSN. pp. 73–84 (2011)
20. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32(4), 824–840 (1985)
21. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. 81, 13–31 (April 1989)
22. Chambart, P., Schnoebelen, P.: Mixing lossy and perfect fifo channels. In: CONCUR. LNCS, vol. 5201, pp. 340–355 (2008)
23. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (March 1996)
24. Charron-Bost, B., Debrat, H., Merz, S.: Formal verification of consensus algorithms tolerating malicious faults. In: SSS (2011)
25. Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice, Lecture Notes in Computer Science, vol. 5959. Springer (2010)
26. Chou, C.T., Mannava, P., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: FMCAD, LNCS, vol. 3312, pp. 382–398. Springer Verlag (2004)

27. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (Sep 2003)
28. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In: TACAS'08/ETAPS'08. pp. 33–47. Springer (2008)
29. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: CONCUR 2004. vol. 3170, pp. 276–291 (2004)
30. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs. LNCS, vol. 131, pp. 52–71 (1981)
31. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
32. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: VMCAI. pp. 126–141 (2006)
33. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
34. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (Oct 1991)
35. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
36. De Prisco, R., Malkhi, D., Reiter, M.K.: On k-set consensus problems in asynchronous systems. IEEE Trans. Parallel Distrib. Syst. 12(1), 7–21 (2001)
37. Dolev, D., Lynch, N.A., Pinter, S.S., Stark, E.W., Weihl, W.E.: Reaching approximate agreement in the presence of faults. J. ACM 33(3), 499–516 (1986)
38. Dutertre, B., De Moura, L.: A fast linear-arithmetic solver for DPLL (T). In: Computer Aided Verification. pp. 81–94. Springer (2006)
39. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (Apr 1988)
40. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: CADE, LNCS, vol. 1831, pp. 236–254. Springer (2000)
41. Emerson, E., Namjoshi, K.: Reasoning about rings. In: POPL. pp. 85–94 (1995)
42. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (Apr 1985)
43. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In: TACAS. LNCS, vol. 4963, pp. 315–331. Springer (2008)
44. Fuegger, M., Schmid, U., Fuchs, G., Kempf, G.: Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In: EDCC-6. pp. 87–96 (Oct 2006)
45. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39, 675–735 (July 1992)
46. Graf, S., Saïdi, H.: Construction of abstract state graphs with pvs. In: CAV. LNCS, vol. 1254, pp. 72–83 (1997)
47. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking - History, Achievements, Perspectives, LNCS, vol. 5000 (2008)
48. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language lustre. IEEE Trans. Softw. Eng. 18, 785–793 (September 1992)
49. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)

50. Ip, C., Dill, D.: Verifying systems with replicated components in murϕ. In: CAV, LNCS, vol. 1102, pp. 147–158. Springer (1996)
51. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Starting a dialog between model checking and fault-tolerant distributed algorithms. arXiv CoRR abs/1210.3839 (2012)
52. 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)
53. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards modeling and model checking fault-tolerant distributed algorithms. In: SPIN. LNCS, vol. 7976, pp. 209–226 (2013)
54. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M.R., Yu, Y.: Checking cache-coherence protocols with TLA$^+$. Formal Methods in System Design 22(2), 125–131 (2003)
55. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science, Morgan&Claypool (2006)
56. Kesten, Y., Pnueli, A.: Control and data abstraction: the cornerstones of practical formal verification. STTT 2, 328–342 (2000)
57. Konnov, I., Veith, H., Widder, J.: Who is afraid of Model Checking Distributed Algorithms? (2012)
58. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16, 133–169 (May 1998)
59. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
60. Lamport, L.: The pluscal algorithm language. In: ICTAC. LNCS, vol. 5684, pp. 36–60 (2009)
61. Lincoln, P., Rushby, J.: A formally verified algorithm for interactive consistency under a hybrid fault model. In: FTCS-23. pp. 402–411 (jun 1993)
62. Lynch, N.: Distributed Algorithms. Morgan Kaufman, San Francisco, USA (1996)
63. Lynch, N., Tuttle, M.: An introduction to input/output automata. Tech. Rep. MIT/LCS/TM-373, Laboratory for Computer Science, MIT (1989)
64. McMillan, K.: Symbolic model checking. Kluwer (1993)
65. McMillan, K.L.: Parameterized verification of the flash cache coherence protocol by compositional model checking. In: CHARME. LNCS, vol. 2144, pp. 179–195 (2001)
66. Mitra, S., Lynch, N.A.: Proving approximate implementations for probabilistic I/O automata. Electr. Notes Theor. Comput. Sci. 174(8), 71–93 (2007)
67. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN. pp. 541–550 (2003)
68. O'Leary, J.W., Talupur, M., Tuttle, M.R.: Protocol verification using flows: An industrial experience. In: FMCAD. pp. 172–179 (2009)
69. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J.ACM 27(2), 228–234 (April 1980)
70. Pnueli, A., Xu, J., Zuck, L.: Liveness with (0,1,∞)- counter abstraction. In: CAV, LNCS, vol. 2404, pp. 93–111. Springer (2002)
71. Powell, D.: Failure mode assumptions and assumption coverage. In: FTCS-22. pp. 386–395. Boston, MA, USA (1992)
72. Santoro, N., Widmayer, P.: Time is not a healer. In: STACS. LNCS, vol. 349, pp. 304–313. Springer (1989)
73. Schmid, U., Weiss, B., Rushby, J.: Formally verified Byzantine agreement in presence of link faults. In: ICDCS. pp. 608–616 (Jul 2-5, 2002)

74. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. Inf. Comput. 206(11), 1313–1333 (2008)
75. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM 34(3), 626–645 (Jul 1987)
76. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing 2, 80–94 (1987)
77. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. 28(4), 213–214 (Jul 1988)
78. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distributed Computing 23(5–6), 341–358 (2011)
79. Widder, J., Biely, M., Gridling, G., Weiss, B., Blanquart, J.P.: Consensus in the presence of mortal Byzantine faulty processes. Distributed Computing 24(6), 299–321 (2012)
80. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distributed Computing 20(2), 115–140 (August 2007)
81. Wöhrle, S., Thomas, W.: Model checking synchronized products of infinite transition systems. LMCS 3(4) (2007)