Reducing Liveness to Safety in First-Order Logic

ODED PADON, Tel Aviv University, Israel
JOCHEN HOENICKE, University of Freiburg, Germany
GIULIANO LOSA, University of California, Los Angeles, USA
ANDREAS PODELSKI, University of Freiburg, Germany
MOOLY SAGIV, Tel Aviv University, Israel
SHARON SHOHAM, Tel Aviv University, Israel

We develop a new technique for verifying temporal properties of infinite-state (distributed) systems. The main idea is to reduce the temporal verification problem to the problem of verifying the safety of infinite-state systems expressed in first-order logic. This allows to leverage existing techniques for safety verification to verify temporal properties of interesting distributed protocols, including some that have not been mechanically verified before. We model infinite-state systems using first-order logic, and use first-order temporal logic (FO-LTL) to specify temporal properties. This general formalism allows to naturally model distributed systems, while supporting both *unbounded-parallelism* (where the system is allowed to dynamically create processes), and infinite-state per process.

The traditional approach for verifying temporal properties of infinite-state systems employs well-founded relations (e.g. using linear arithmetic ranking functions). In contrast, our approach is based the idea of fair cycle detection. In finite-state systems, temporal verification can always be reduced to fair cycle detection (a system contains a fair cycle if it revisits a state after satisfying all fairness constraints). However, with both infinitely many states and infinitely many fairness constraints, a straightforward reduction to fair cycle detection is unsound. To regain soundness, we augment the infinite-state transition system by a dynamically computed finite set, that exploits the locality of transitions. This set lets us define a form of fair cycle detection that is sound in the presence of both infinitely many states, and infinitely many fairness constraints. Our approach allows a new style of temporal verification that does not explicitly involve ranking functions. This fits well with pure first-order verification which does not explicitly reason about numerical values. In particular, it can be used with effectively propositional first-order logic (EPR), in which case checking verification conditions is decidable. We applied our technique to verify temporal properties of several interesting protocols. To the best of our knowledge, we have obtained the first mechanized liveness proof for both TLB Shootdown, and Stoppable Paxos.

CCS Concepts: • **Theory of computation → Logic and verification**; *Modal and temporal logics*; • **Software and its engineering → Formal methods**;

Additional Key Words and Phrases: Liveness-to-safety reduction, first-order temporal logic

ACM Reference Format:
Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing Liveness to Safety in First-Order Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 26 (January 2018), 33 pages. https://doi.org/10.1145/3158114

Authors' addresses: Oded Padon, Tel Aviv University, Israel, odedp@mail.tau.ac.il; Jochen Hoenicke, University of Freiburg, Germany, hoenicke@informatik.uni-freiburg.de; Giuliano Losa, University of California, Los Angeles, USA, giuliano@cs.ucla.edu; Andreas Podelski, University of Freiburg, Germany, podelski@informatik.uni-freiburg.de; Mooly Sagiv, Tel Aviv University, Israel, msagiv@post.tau.ac.il; Sharon Shoham, Tel Aviv University, Israel, sharon.shoham@gmail.com.

2475-1421/2018/1-ART26
https://doi.org/10.1145/3158114

1 INTRODUCTION

This paper is motivated by the problem of verifying liveness properties of distributed protocols with unbounded resources and dynamic creations of objects. We are interested in handling both *unbounded-parallelism* (where the system is allowed to dynamically create processes), and *infinite-state per process.*

Technically, we reduce the temporal verification problem to the problem of verifying the safety of infinite-state systems expressed in pure first-order logic. This allows to leverage existing techniques for safety verification to verify temporal properties of interesting distributed protocols, including some that have not been mechanically verified before.

Modeling in first-order logic. We model infinite-state systems using first-order logic, and use first-order temporal logic (FO-LTL) (see e.g., [Abadi 1989; Manna and Pnueli 1983]) to specify temporal properties. This general formalism provides a powerful and natural way to model many infinite-state systems. It is particularly natural for distributed and multi-threaded systems, where quantifiers can be used to reason about the multiple nodes or threads, as well as messages, values, and other objects of the system, while supporting both unbounded-parallelism and infinite-state per process. The states of the system are modeled as first-order logic structures (rather than, say, fixed-length tuples of integer values), which allows a rich mechanism for formalizing various aspects of the state, such as dynamic sets of threads and processors, message channels, and unbounded local and global state. Such models can also account for counters, i.e., variables over the natural numbers with operations such as increment, decrement, and comparison. For example, a mutual exclusion protocol such as the ticket protocol uses two counters to implement a waiting queue in an unbounded array.

The fact that we need to go beyond fixed-length tuples of values to model a state and use first-order logic structures (with unboundedly large or even infinite domains) may seem like an extra burden on the task of proving liveness. We will show that, on the contrary, the formalism of first-order logic gives us a unique opportunity for proving liveness in a new manner.

Several previous works (e.g., [McMillan 2016; Padon et al. 2017, 2016]) demonstrated the utility of the first-order formalism for deductive verification of safety properties using inductive invariants in first-order logic. The reduction presented in this work opens the path to also use first-order logic to verify temporal properties. By applying the liveness-to-safety reduction we develop, one can prove liveness, and in fact arbitrary temporal properties, using the well-investigated concept of inductive invariants.

Liveness properties. A liveness property specifies that *a good thing will happen.* For example, non-starvation for a mutual-exclusion protocol specifies that every request to access some critical resource will eventually be granted. It is typical of a liveness property that its validity depends on fairness assumptions (e.g., fair thread scheduling, eventual message delivery). For infinite-state systems, infinitely many fairness constraints may be needed (e.g., for unbounded-parallelism). A counterexample to a liveness property is an *infinite* fair execution, satisfying (possibly infinitely many) fairness constraints. Reasoning about liveness and fairness properties for a distributed protocol or any other concurrent program with a parameterized or dynamic number of threads and/or dynamic data structures is notoriously difficult.

To see why, consider a distributed protocol where the maximal number of steps until the *good thing* happens depends on the success of actions on a set of objects (threads, tasks in a list, nodes in a graph, messages in a queue, etc.). The set of objects may change dynamically and grow indefinitely through a sequence of interleaving actions (thread creation, etc.). In fact, in an infinite trace, this set itself can be infinite. As an example, consider a bug that causes newly created threads to become first-in-line for access to a critical resource. Then, a thread can become starved if new threads keep being created ahead of it. In the infinite counterexample trace, there is an infinite number of threads. Liveness proofs must therefore take into account both the control flow and infinitely many fairness assumptions to determine the possible sequences and prove that their interleaving actions cannot stall the progress towards the *good thing* forever.

Our approach: liveness-to-safety reduction. In general, liveness-to-safety reductions are a recent theme in research on methods to prove program termination and liveness properties (see the discussion of related work in Section 8). The rationale is that we have an ever growing arsenal of scalable techniques to automatically synthesize inductive invariants but we have only limited ways to synthesize ranking functions. Thus the goal is to shift as much burden as possible from the task of finding ranking functions to the task of finding inductive invariants. The problem is exacerbated by the fact that techniques to synthesize ranking functions apply mostly only to a limited range of data structures (essentially, linear arithmetic). Our contribution is a liveness-to-safety reduction that gets rid of the task of finding ranking functions altogether.

For finite-state systems, liveness can be proven through acyclicity (the absence of fair cycles in every executions). This is the classical liveness-to-safety reduction, a term coined in [Biere et al. 2002]. This also works for parameterized systems, where the state-space is infinite, but actually finite (albeit unbounded) for every system instance [Pnueli and Shahar 2000]. It is well-known that, to prove a liveness property of an infinite-state system, an argument based on acyclicity would be unsound (an infinite-state system can be acyclic but non-terminating). We will show that, when we use a first-order logic to formalize *first-order fair transition systems*, there is a canonical way to derive an abstract semantics for which a suitable acyclicity test is sound.

While it is sound to test acyclicity on a finite-state system resulting from an abstraction, it is also void because in general, an abstraction that maps an infinite set of states to a finite set will introduce cycles in the resulting finite-state system (even if there were no cycles before). We avoid this by fine-tuning the abstraction individually for each execution, while abstracting only the cycle detection aspect (rather than the actual transitions of the system). Such fine-tuned abstraction is possible using the symbolic representation of the transition relation in first-order logic, as well as the first-order formulation of the fairness constraints.

The basic observation which we use here is that, once a finite domain of objects is fixed, there exist only finitely many first-order logic structures over the same signature, providing a natural finite abstraction by projection. To determine *how* to fix the finite domain of objects, we note that, an execution can be a counterexample only if it satisfies *all* fairness assumptions. However, to prove that a set of executions satisfies the given liveness property, in general we need only a finite number of fairness assumptions in any point in time. The key idea here is that the finite set of needed fairness assumptions can be selected to fix a finite domain of objects, and vice versa.

Main results. The contributions of this paper can be summarized as follows:

- We define a parameterized fair cycle detection mechanism that is sound for proving fair termination of transition systems with both infinitely many states and infinitely many fairness constraints.
- We instantiate the parameterized mechanism in a uniform way for transition systems expressed in first-order logic, exploiting the *footprint* of transitions. For such systems, we obtain an *algorithmic* reduction from verification of arbitrary temporal properties to verification of safety properties.
- We extend the applicability of the reduction by allowing a user to specify a nesting structure which breaks the termination argument into levels.
- We demonstrate the utility of our approach by applying it to verify liveness properties of several interesting protocols: ticket protocol (with unbounded-parallelism), alternating

bit protocol, TLB shootdown protocol, and three variants of Paxos, including Stoppable Paxos. Our evaluation indicates that in many cases the safety problem obtained from the reduction can be verified using verification conditions in first-order logic, and specifically in the decidable EPR fragment.
- To the best of our knowledge, we provide the first mechanized liveness proof for both TLB Shootdown and Stoppable Paxos. Interestingly, Stoppable Paxos is tricky enough that [Lamport et al. 2008] prove its liveness using an informal proof of about 3 pages of temporal-logic reasoning.

2 OVERVIEW

In this section we present the main ideas of our approach for proving temporal properties of infinite-state systems using first-order logic, enabled by our novel liveness-to-safety reduction.

Section 2.1 introduces a running example that we use to illustrate our approach. Section 2.2 shows how to specify infinite-state systems and their temporal properties using first-order temporal logic (FO-LTL), a well established combination of pure (uninterpreted) first-order logic and linear temporal logic (see e.g., [Manna and Pnueli 1995]). Our approach for temporal verification of such systems consists of two steps (reductions), summarized in Fig. 1. In the first step, we reduce verification of temporal properties to verification of fair termination, where the transition system and the fairness constraints are specified in first-order logic (this is in resemblance to the finite state case). In the second step, we reduce the problem of fair termination of a first-order transition system, to a safety verification problem for a first-order transition system. The latter reduction is the main contribution of the paper, sketched in Section 2.3.

What we gain. Once the temporal verification is reduced to safety verification of a first-order transition system, the safety property can be semi-automatically proven by supplying an inductive invariant (in first-order logic), and then proving the resulting verification conditions using first-order theorem provers. Furthermore, as our examples show, in many cases the resulting verification conditions are in the decidable EPR fragment [Piskac et al. 2010; Ramsey 1930].

Reducing temporal verification to verification conditions in first-order logic (and EPR in particular) has both theoretical and practical benefits. Theoretically, in contrast to more powerful logics, first-order logic has a complete proof system. Practically, great progress has been made in automated first-order theorem proving (e.g., SPASS [Weidenbach et al. 2009], Vampire [Riazanov and Voronkov 2002], iProver [Korovin 2008]), including support for EPR. Our approach allows to leverage this vast progress for temporal verification.

2.1 A Running Example

We illustrate our approach using the *ticket protocol* for ensuring mutual exclusion with non-starvation among multiple threads, depicted in Fig. 2. The ticket protocol (a variant of Lamport's bakery algorithm [Lamport 1974]) is an idealized version of spinlocks used in the Linux kernel [Corbet 2008]. The protocol uses natural numbers as ticket values. The global state contains a variable, n, that records the next available ticket, and a variable, s, that records the ticket that is currently being served. Each thread contains a local ticket variable m. Each thread that wishes to enter the critical section runs the code depicted in Fig. 2, where it first acquires a ticket by setting a local variable m to n and atomically incrementing the next available ticket n. It then waits until its ticket m is equal to s, the ticket that is served. When this happens, it enters the critical section. When it exits the critical section, it increases s, allowing the next thread to be served.

We note that the ticket protocol may be run by any number of threads. In fact, the ticket protocol supports the *unbounded-parallelism* model, in which new threads may be created during the run of

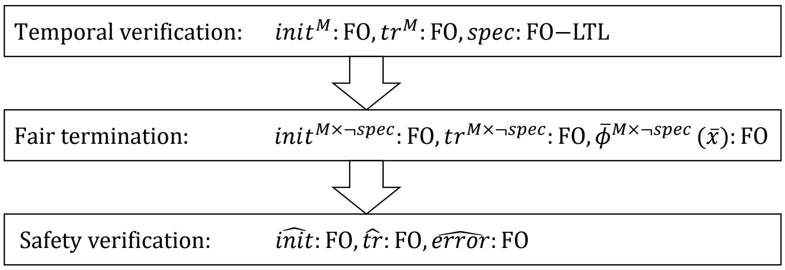

Temporal verification: $init^M$: FO, tr^M: FO, $spec$: FO−LTL
Fair termination: $init^{M\times\neg spec}$: FO, $tr^{M\times\neg spec}$: FO, $\bar{\phi}^{M\times\neg spec}(\bar{x})$: FO
Safety verification: $\widehat{init}$: FO, $\widehat{tr}$: FO, $\widehat{error}$: FO

Fig. 1. Flow of verification of temporal properties of infinite state systems in first-order logic. The input model is given by a first-order specification of the initial states and transition relation, and by a temporal specification in FO-LTL. It is then transformed to a fair transition system (fully specified in first-order logic, without FO-LTL), whose fair traces are exactly traces of the original model that violate its temporal specification. The main contribution described in this paper is the reduction from this fair transition system to a safety problem, by constructing a new transition system (specified in first-order logic) such that if it does not reach its error state, then the input model satisfies its temporal specification.

```
    global nat s, n
    local nat m
1:  while (true) {
      m=n++; // Acquire a ticket
2:    while (m>s) { // Busy wait
        skip;
      }
      // Critical section
3:    s++; // Exit critical
    }
```

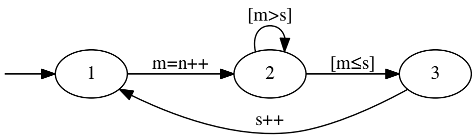


Fig. 2. The ticket protocol for mutual exclusion. Threads in state 1 are idle; a thread in state 2 is waiting to enter the critical section; and state 3 is the critical section.

the protocol. Thus, when considering infinite traces of the ticket protocol, we cannot assume a finite set of threads, not even an unbounded one. Similarly, the set of ticket numbers is infinite. The latter is true even if we assume that the number of threads is finite, since threads may attempt to enter the critical section infinitely often.

2.2 First Order Temporal Specification

The standard way to formalize verification problems is by a triple $(init^M, tr^M, spec)$, where $init^M$ is a formula that describes the set of initial states of a model M, tr^M is a two vocabulary formula that describes M's transition relation, and *spec* is a formula that describes the specification we wish to verify for M. In this paper, we suggest to verify temporal properties of infinite-state systems by using pure (uninterpreted) first-order logic for $init^M$ and tr^M, and FO-LTL for *spec*. Notice that the FO-LTL specification goes beyond pure first-order logic, since the semantics of linear temporal logic (where time ranges over the natural numbers) is not first-order expressible (see e.g., [Abadi 1989]).

We now illustrate this formalism using the ticket example.

Transition system in first-order logic. In our running example, the formulas for $init^M$ and tr^M use first-order logic to implement Fig. 2. They are rather straightforward, and appear in full detail later in the paper (see Fig. 4). The only difference between the natural specification and ours is that we model ticket numbers using a general total order (axiomatizable in first-order logic), instead of using the natural numbers (which are not axiomatizable in first-order logic). This amounts to

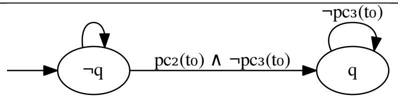
Liveness Property	$\forall x : \text{thread}.\ \square\, (pc_2(x) \rightarrow \lozenge\, pc_3(x))$
Fairness Assumption	$\forall x : \text{thread}.\ \square \lozenge\, scheduled(x)$
Temporal Spec. (*spec*)	$(\forall x : \text{thread}.\ \square \lozenge\, scheduled(x)) \rightarrow \forall x : \text{thread}.\ \square\, (pc_2(x) \rightarrow \lozenge\, pc_3(x))$
$\neg spec$	$(\forall x : \text{thread}.\ \square \lozenge\, scheduled(x)) \wedge (\exists x : \text{thread}.\ \lozenge\, (pc_2(x) \wedge \square \neg pc_3(x)))$
$Skolem(\neg spec)$	$(\forall x : \text{thread}.\ \square \lozenge\, scheduled(x)) \wedge \lozenge\, (pc_2(t_0) \wedge \square \neg pc_3(t_0))$
$T_{Skolem(\neg spec)}$	 $\phi_1(x) = scheduled(x)$ $\phi_2 = q$

Fig. 3. Expressing the temporal specification of the ticket protocol. The liveness property and fairness assumptions are expressed in FO-LTL. The FO-LTL specification of the protocol states that the fairness assumptions imply the liveness properties. To obtain a fair transition system that captures infinite traces that violate the specification, we first negate the specification and Skolemize the result (t_0 is a Skolem constant from the subformula $\exists x : \text{thread}.\ \lozenge(pc_2(x) \wedge \square \neg pc_3(x))$). We then convert it to the depicted fair transition system $T_{Skolem(\neg spec)}$. $T_{Skolem(\neg spec)}$ has two states, labeled q and $\neg q$ (q is a new nullary relation), and fairness constraints $\phi_1(x)$, ϕ_2.

a sound abstraction, which is incomplete in general. However, in the examples we consider, this abstraction does not hurt us.

Temporal specification. Fig. 3 depicts the FO-LTL specification for the ticket protocol. The specification we wish to verify is that every thread that requests to enter the critical section (i.e., reaches location 2), eventually enters (i.e., reaches location 3). This should hold under the fairness assumption that all the threads are scheduled infinitely often. Thus, the FO-LTL specification is that the fairness assumption implies liveness. Fig. 3 also depicts the negation of the specification, and a "monitor" that tracks it, which we explain next.

Reduction from FO-LTL specification to fair termination. As a first step in our approach for verifying that a transition system given by $(init^M, tr^M)$ in first-order logic satisfies a specification given by an FO-LTL formula *spec*, we follow an approach which generalizes the standard automata theoretic approach from the propositional case to the first-order case. Specifically, we reduce the problem of verifying a temporal specification for a transition system, to the problem of checking that a *fair transition system* has no infinite fair traces.

A *fair transition system* is specified by $(init, tr, \{\phi_1(\bar{x}), \ldots, \phi_n(\bar{x})\})$, where *init* and *tr* specify initial states and transitions, and each $\phi_i(\bar{x})$ is a first-order formula with free variables, that specifies Büchi acceptance conditions, which we call fairness constraints. The semantics of the fairness constraints requires that a fair trace must satisfy each fairness constraint infinitely often, for *every* assignment to the free variables. They thus capture *infinitely many* fairness constraints, for example, that every thread is scheduled infinitely often.

To reduce the problem of checking if $(init^M, tr^M) \models spec$, we first negate *spec* and Skolemize it. We then convert the negation of the specification to a fair transition system, whose fairness constraints will be derived both from the original fairness assumptions, and from the negation of the liveness property. This process is illustrated for the ticket example in Fig. 3. We then take the product of the original transition system $(init^M, tr^M)$, and the transition system that encodes the negation of *spec*. The result is a fair transition system, whose fair traces are in one-to-one correspondence to runs of the $(init^M, tr^M)$ that violate *spec*. From this point, all that remains is to check if the resulting fair transition system is empty or not (i.e., does it have a fair trace). This reduction to fair termination is both sound and complete.

For the ticket example, the result is a fair transition system with an additional Skolem constant t_0 and an additional nullary relation q, which represents the state of an automaton that is depicted in Fig. 3, that encodes the negation of the liveness property. The resulting transition system has two fairness constraints: $\phi_1 = scheduled(x)$, $\phi_2 = q$. Note that the first constraint has a free thread variable, in order to enforce that every thread is scheduled infinitely often.

2.3 Reducing Fair Termination to Safety

The core result in this paper is a technique for reducing the fair termination problem of a fair transition system $(init, tr, \{\phi_1(\bar{x}), \ldots, \phi_n(\bar{x})\})$ to a safety problem given by $(\widehat{init}, \widehat{tr}, \widehat{error})$, when both systems are specified in first-order logic. This allows standard methods for proving safety to be used, e.g., deductive verification and abstract interpretation.

2.3.1 Intuition from the Ticket Protocol Example. Before presenting our reduction, let us gain some intuition from the example of the ticket protocol. Consider the following intuitive argument that explains why the ticket protocol satisfies its specification. We need to show that there is no infinite trace in which every thread is scheduled infinitely often, but at some point it time, say k_0, thread t_0 takes a ticket (enters pc_2) and from k_0 on, t_0 never enters the critical section (pc_3). We observe that to show that no such trace exists, it suffices to consider the finite set A of threads and ticket numbers that are active at k_0, that is, all threads that were scheduled prior to k_0, and all ticket numbers allocated prior to k_0. The reason is that for any interval $[k_1, k_2]$ later than k_0 in which all the threads in A are scheduled, *the state of at least one of them changed*, when restricted to the allocated tickets in A. This is because one of these threads is the one being serviced. In other words, when projecting the states of the protocol to the finite set A, there is no abstract cycle that visits all the fairness constraints induced by A. This resembles the liveness-to-safety reduction of [Biere et al. 2002] in the case of finite-state systems. Next, we present our reduction and discuss this relation in more detail.

2.3.2 Reducing Fair Termination to Safety. The essence of our reduction from fair termination to safety is to identify a family of finite execution traces, such that every fair trace must contain one of them as a prefix. After such a family is identified, we are left with the safety problem of showing that no such finite trace is reachable. In the finite-state case, the classical reduction of [Biere et al. 2002] uses for this purpose the family of *fair cycles* — lasso shaped finite executions that visit the same state twice, while visiting every fairness constraint at least once in the loop.

A naive use of the family of fair cycles fails for infinite-state systems, and for two reasons. First, with infinitely many states, a trace can be infinite without repeating the same state twice. Second, with infinitely many fairness constraints, the condition that a finite path visits all fairness constraints is inadequate: an infinite trace may visit every fairness constraint infinitely often, but without having any finite segment that visits all of them even once.

When considering fair transition systems specified in first-order logic, both problems can be eliminated if we consider a *finite* subset D of elements. We can then project the states of the system into D, and obtain a *finite abstraction* of the state space, such that any infinite (concrete) trace will revisit some *abstract state* infinitely often. Furthermore, we can use D to define a finite set of fairness constraints F_D, which we obtain by instantiating the fairness constraints $\phi(x)$ only with elements from D. For the ticket example, this means we would only require that every thread is scheduled for a finite set of threads. Now, in every infinite fair trace, there must be a D-abstract fair cycle, i.e., an infinite fair trace must visit two states s_1 and s_2 such that $s_1|_D = s_2|_D$, and every fairness constraint in F_D is visited by the path from s_1 to s_2. This characterization, combined with a mechanism for computing D described next, provides the desired family of finite execution traces, such that every fair trace must contain one of them as a prefix.

Consider the ticket example. We see that the above argument captures the essence of the intuitive proof we presented, provided that the set D contains all threads active at k_0, i.e., the time where t_0 obtains its ticket number, and all ticket numbers allocated at that time. Indeed, for the ticket example, as well as for other interesting examples, one cannot use any *a priori* fixed finite set D. A key enabling insight we present is to determine a different D for different traces, in a way which ensures that for any fair trace, we would assign a finite set D. In the ticket example, we can define D to be the set of threads scheduled and ticket numbers allocated prior to k_0. A key question is how to determine D in the general case.

Dynamic abstraction and fairness selection using footprint. For determining D, we present a natural solution that is sufficient to fully automate the reduction (i.e., we do not ask for the user's help), while being precise enough to capture challenging examples. Our solution is to maintain a finite set of elements that is the *footprint* of a finite execution prefix. Transitions are usually *local*, in the sense that each transition affects (and is affected by) a finite set of elements. We show that we can syntactically extract this finite set from tr^M. For the ticket example, the footprint of a prefix would precisely be all the threads scheduled, and all the ticket numbers allocated.

This gives rise to a natural way to define D. Given a trace, wait for a time k_0 when a certain condition is met, and then let D be the footprint of the execution prefix up to k_0. In the ticket example, a suitable condition is that thread t_0 obtains its ticket number. In the general case, we must choose a condition such that every infinite fair trace eventually meets it. Our canonical solution for this is to wait for a finite subset of the fairness constraints, which is determined by the initial state. In particular, it will include all nullary fairness constraints, i.e., fairness constraints given by formulas without free variables. For the ticket example, $\phi_2 = q$ is a nullary fairness constraint, ensuring that k_0 is after t_0 obtains its ticket number. This works well for other interesting examples as well.

Realization of the reduction. With these definitions of D and k_0, we define an algorithmic reduction of fair termination to safety in first-order logic. Given a fair first-order transition system, the reduction augments it by a monitor that identifies finite execution traces that (1) reach the point k_0 in which D is set to the footprint of the execution so far, and (2) afterwards visit two states s_1 and s_2 such that $s_1|_D = s_2|_D$ and every fairness constraint in F_D is visited by the path from s_1 to s_2. Upon identifying such a prefix, the monitor enters an error state. The output of the reduction is the augmented transition system and the safety property that the error state is not reachable.

Note that, while the soundness of our reduction relies on a notion of a dynamic finite abstraction, we do not construct an abstract transition system. In fact, the reduction does not incur any abstraction of the transitions. The finite abstraction is only used by the monitor that augments the transition system, for abstract fair cycle detection.

Parameterized systems. Parameterized systems are a special case of infinite-state systems, where each value of the parameter defines a finite-state instance of the system, but the parameter itself is unbounded. In first-order logic, this means we are only interested in traces over finite domains. In this case, it is sound to prove fair termination by proving the absence of fair cycles. This can be viewed as a special case of our approach, where the footprint includes all elements of the domain. Note that for this setting, the reduction to safety is complete, i.e., if the original transition system has no infinite fair traces, then the resulting system will be safe. While our main interest here is in systems that are truly infinite-state, it is nice to note that our general formalism maintains completeness in the special case of parameterized systems.

2.4 A Nested Termination Argument

For finite-state systems and for parameterized transition systems the reduction from fair termination to fair cycle detection is complete. For infinite-state systems the reduction we present is sound but not complete. This is expected, since fair termination is theoretically harder than safety[1]. Incompleteness of the reduction means that for some instances where the input system has no fair traces, the output system will not be safe. While incompleteness is unavoidable, we would like a reduction that works for examples of interest. However, the reduction presented in the previous section fails for a particular class of natural examples.

Intuitively, this happens when progress is not obtained in a bounded number of steps after we fix the finite set of elements used for abstract fair cycle detection. To handle such examples, we develop a more powerful reduction that relies on a *nesting structure* provided by the user. A nesting structure divides the transitions of a system into several nested levels. Given such a structure, one can use abstract fair cycle detection for each level separately, while assuming subsequent levels terminate.

3 FIRST ORDER SPECIFICATION OF INFINITE-STATE SYSTEMS

In this section, we present our formalism for specifying infinite-state systems and their properties.

3.1 Transition Systems

A *transition system* is a tuple (S, S_0, R), where S is a (possibly infinite) state-space, $S_0 \subseteq S$ is the set of *initial states* and $R \subseteq S \times S$ is the *transition relation.* A (finite or infinite) trace is a sequence of states $\pi = s_0, s_1, \ldots$ where $s_0 \in S_0$ and $(s_i, s_{i+1}) \in R$ for every $0 \leq i < |\pi|$. We sometimes denote an infinite trace by $(s_i)_{i=0}^{\infty}$.

Safety. A *safety property* is specified by a set $P \subseteq S$ of "good" states. A transition system (S, S_0, R) satisfies the safety property P if all the reachable states are in P, where the set of reachable states is defined in the usual way as the set of states along any trace.

Fairness. A (Büchi) *fair transition system* is a tuple $(S, S_0, R, \mathcal{F})$ where S, S_0, R are defined as before and $\mathcal{F} \subseteq \mathcal{P}(S)$ is a (possibly infinite) set of *fairness constraints.* An infinite trace $\pi = s_0, s_1, \ldots$ is *fair* if it visits every fairness constraint $F \in \mathcal{F}$ infinitely often, i.e., the sets $\{i \mid s_i \in F\}$ are infinite.

Fair termination. The transition system $(S, S_0, R, \mathcal{F})$ *terminates* if it has no fair (infinite) traces.

3.2 Transition Systems in First-Order Logic

We now provide a formalism for specifying transition systems in first-order logic. We note that this formalism is Turing-complete. Furthermore, existing tools, such as IVy [Padon et al. 2016], provide modeling languages that are closer to imperative programming languages and compile to a first-order transition system. This makes it easier for a user to provide a first-order specification of the transition system they wish to verify.

Syntax. A first-order logic specification of a transition system is $(\Sigma, \varphi_0, \tau)$, where Σ is a first-order vocabulary. We assume Σ contains only relation symbols and constant symbols (functions can be

[1]For transition systems in first-order logic, it is easy to show that fair termination is Π_1^1 hard (see [Abadi 1989]), while safety is in the arithmetical hierarchy.

$$\Sigma^t = \{n : \text{ticket}, s : \text{ticket}, m : \text{thread} \to \text{ticket}, \leq (\text{ticket}, \text{ticket}), pc_1(\text{thread}), pc_2(\text{thread}), pc_3(\text{thread}), \mathit{scheduled}(\text{thread})\}$$
$$\varphi_0^t = \psi_{\min}(n) \land \psi_{\min}(s) \land \forall x : \text{thread}.\ pc_1(x) \land \neg pc_2(x) \land \neg pc_3(x) \land \psi_{\min}(m(x)) \land \neg \mathit{scheduled}(x)$$
$$\tau^t = \exists x : \text{thread}.\ \tilde{\tau}(x)$$
$$\tilde{\tau}(x) = (\tau_{12}(x) \lor \tau_{22}(x) \lor \tau_{23}(x) \lor \tau_{31}(x)) \land \forall y : \text{thread}. \mathit{scheduled}'(y) \leftrightarrow x = y$$
$$\tau_{12}(x) = \psi_{12}(x) \land m'(x) = n \land (\forall y : \text{thread}.\ x \neq y \to m'(y) = m(y)) \land \psi_{\text{succ}}(n, n') \land s' = s$$
$$\tau_{22}(x) = \psi_{22}(x) \land m(x) > s \land (\forall y : \text{thread}.\ m'(y) = m(y)) \land n' = n \land s' = s$$
$$\tau_{23}(x) = \psi_{23}(x) \land m(x) \leq s \land (\forall y : \text{thread}.\ m'(y) = m(y)) \land n' = n \land s' = s$$
$$\tau_{31}(x) = \psi_{31}(x) \land (\forall y : \text{thread}.\ m'(y) = m(y)) \land n' = n \land \psi_{\text{succ}}(s, s')$$
$$\psi_{ij}(x) = pc_i(x) \land pc'_j(x) \land \left(\bigwedge_{k \neq j} \neg pc'_k(x)\right) \land \bigwedge_k \forall y : \text{thread}.\ x \neq y \to (pc'_k(y) \leftrightarrow pc_k(y))$$
$$\psi_{\min}(x) = \forall y : \text{ticket}.\ x \leq y$$
$$\psi_{\text{succ}}(x, y) = x < y \land \forall z : \text{ticket}.\ x < z \to y \leq z$$
$$\psi_{\text{total order}} = (\forall x : \text{ticket}.\ x \leq x) \land (\forall x, y, z : \text{ticket}.\ x \leq y \land y \leq z \to x \leq z) \land$$
$$(\forall x, y : \text{ticket}.\ x \leq y \land y \leq x \to x = y) \land (\forall x, y : \text{ticket}.\ x \leq y \lor y \leq x)$$

Fig. 4. First-order logic specification of the ticket protocol. The vocabulary includes uninterpreted relations and functions, and a total order on ticket values $\leq$, axiomatized by $\psi_{\text{total order}}$.

encoded by relations). φ_0 is a closed formula over Σ and τ is a closed formula over $\Sigma \uplus \Sigma'$, where $\Sigma' = \{r' \mid r \in \Sigma\}$. Fig. 4 depicts the first-order logic specification of the ticket protocol.

Semantics. A first-order specification $(\Sigma, \varphi_0, \tau)$ defines a class of transition systems, one for each domain $\mathcal{D}$. Let $\mathcal{D}$ be any set (possibly infinite), then the transition system (S, S_0, R) defined by $(\Sigma, \varphi_0, \tau)$ is given by:

$$S = \{s = (\mathcal{D}, \mathcal{I}) \mid \mathcal{I} \text{ is an interpretation of } \Sigma \text{ for domain } \mathcal{D}\}$$
$$S_0 = \{s \in S \mid s \models \varphi_0\} \qquad R = \{(s, s') \in S \times S \mid (s, s') \models \tau\}$$

In the above definition, given $s = (\mathcal{D}, \mathcal{I}) \in S$ and $s' = (\mathcal{D}, \mathcal{I}') \in S$, we use (s, s') as a shorthand for the structure $(\mathcal{D}, \mathcal{I} \uplus \mathcal{I}'')$, where $\mathcal{I}'' = \lambda r' \in \Sigma'. \mathcal{I}'(r)$. Namely, the structure defined by (s, s') is a structure over the vocabulary $\Sigma \uplus \Sigma'$ with the same domain as s and s', and where the symbols in Σ are interpreted as in s, and the symbols in Σ' are interpreted as in s'.

A *trace* of $(\Sigma, \varphi_0, \tau)$ is a trace of the transition system (S, S_0, R) for some $\mathcal{D}$. As such, a trace is a sequence of first-order structures over Σ. Every state along the trace has its own interpretation of the constant and relation symbols, but they all share the same domain $\mathcal{D}$. The reachable states of $(\Sigma, \varphi_0, \tau)$ consist of all the states reachable in (S, S_0, R) for some $\mathcal{D}$.

Safety. A *safety property* for a first-order logic transition system $(\Sigma, \varphi_0, \tau)$ is specified via a closed first-order logic formula φ_P over Σ. $(\Sigma, \varphi_0, \tau)$ satisfies φ_P if all the reachable states satisfy φ_P.

Inductive invariants. A prominent way for proving safety properties uses inductive invariants. An inductive invariant for a first-order logic transition system $(\Sigma, \varphi_0, \tau)$ and a safety property φ_P is a closed formula I over Σ such that: $\varphi_0 \to I$, $I \land \tau \to I'$ and $I \to \varphi_P$ are all valid.

Fairness. A first-order logic specification of a fair transition system is $(\Sigma, \varphi_0, \tau, \Phi)$, where Σ, φ_0, τ are as before and $\Phi = \{\phi_1, \ldots, \phi_n\}$, where each ϕ_i is a formula over Σ that may contain free variables. For each domain $\mathcal{D}$, the semantics of $(\Sigma, \varphi_0, \tau, \Phi)$ is a fair transition system $(S, S_0, R, \mathcal{F})$

where S, S_0, R are as before and $\mathcal{F}$ is a possibly infinite set of fairness constraints defined by Φ:

$$\mathcal{F} = \left\{ F_\phi(\bar{e}) \mid \phi(x_1, \ldots, x_n) \in \Phi,\ \bar{e} \in \mathcal{D}^n \right\}$$
$$\text{where } F_\phi(\bar{e}) = \{ s \in S \mid s, [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] \models \phi(x_1, \ldots, x_n) \}$$

Note that each fairness formula $\phi \in \Phi$ with free variables induces a set of fairness constraints of the form $F_\phi(\bar{e})$, for each $\bar{e} \in \mathcal{D}^n$. Observe that since $\mathcal{D}$ may be infinite, ϕ may stand for infinitely many such fairness constraints.

Fair termination. The fair transition system $(\Sigma, \varphi_0, \tau, \Phi)$ *terminates* if it has no fair (infinite) traces, i.e., if for any $\mathcal{D}$, the transition system defined by $\mathcal{D}$ terminates.

3.3 First-Order Linear Temporal Logic (FO-LTL)

To specify temporal properties of first-order transition systems we use First-Order Linear Temporal Logic (FO-LTL), which combines LTL with first-order logic.

Syntax. Given a first-order vocabulary Σ, FO-LTL formulas are defined by:

$$f ::= r(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg f \mid f_1 \vee f_2 \mid \exists x. f \mid \forall x. f \mid \bigcirc f \mid f_1 U f_2$$

where r is an n-ary relation symbol in Σ, each t_i is a term over Σ (defined as in first-order logic), $\bigcirc$ denotes the "next" temporal operator and U denotes the "until" temporal operator. We also use the standard shorthands for the "eventually" and "globally" temporal operators: $\Diamond f = \mathit{true}\, U f$ and $\Box f = \neg \Diamond \neg f$.

Semantics. FO-LTL formulas are interpreted over infinite traces of a first-order transition system $(\Sigma, \varphi_0, \tau)$ (with the same vocabulary)[2]. Atomic formulas are interpreted over states (which are first-order structures), the temporal operators are interpreted as in traditional LTL, and first-order quantifiers are interpreted over the shared domain $\mathcal{D}$ of all states in the trace. For a formal definition of FO-LTL semantics see [Abadi 1989].

A transition system $(\Sigma, \varphi_0, \tau)$ satisfies an FO-LTL formula f over Σ if all of its traces satisfy f.

3.4 Reducing FO-LTL Verification to Fair Termination

Given any closed formula f in FO-LTL over vocabulary Σ, we can construct a fair transition system (monitor) over an extended vocabulary $\Sigma^f \supseteq \Sigma$ such that the pointwise-projection of its set of fair traces on Σ is exactly the set of all traces that satisfy f. This is a straightforward extension of the classical construction of a Büchi automaton for an LTL formula used in the automata theoretic approach to verification [Vardi and Wolper 1986; Wolper 2000], except that instead of a finite-state automaton we obtain a first-order transition system.

To check whether a given transition system $(\Sigma, \varphi_0, \tau)$ satisfies f we can then take the product of the "monitor" of $\neg f$ and the original transition system. Proving that the original transition system satisfies the FO-LTL formula f reduces to proving that the product transition system has no fair traces. This reduction is sound and complete, meaning that $(\Sigma, \varphi_0, \tau)$ satisfies f if and only if the fair transition system terminates. As such, from now on we focus on proving fair termination. This allows us to verify arbitrary FO-LTL properties.

Example 3.1. Consider the first-order logic specification of the ticket protocol, $(\Sigma^t, \varphi_0^t, \tau^t)$, presented in Fig. 4. Its desired FO-LTL specification, and the steps of constructing a monitor for its negation are presented in Fig. 3. The constructed monitor is a fair transition system, defined over vocabulary $\{pc_2(\text{thread}),\ pc_3(\text{thread}),\ \mathit{scheduled}(\text{thread}),\ t_0 : \text{thread},\ q\}$. We compose it with the

[2]We interpret FO-LTL formulas over transition systems without fairness since fairness can be specified as part of the FO-LTL formula: all fair traces of $(\Sigma, \varphi_0, \tau, \Phi)$ satisfy $f \in$ FO-LTL iff all traces of $(\Sigma, \varphi_0, \tau)$ satisfy $(\bigwedge_{\phi \in \Phi} \forall \bar{x}. \Box \Diamond \phi(\bar{x})) \to f$.

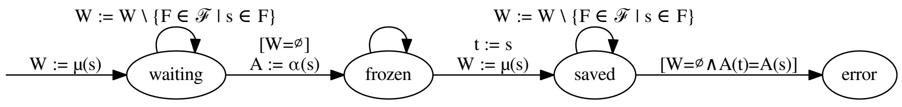


Fig. 5. Monitor that checks the (α,μ)-acyclicity condition. The monitor uses auxiliary state: $W \in \mathcal{P}_{fin}(\mathcal{F})$, $A : S \to X$, and $t \in S$. The variable s denotes the current state of the monitored transition system. W is a finite set of fairness constraints for which the monitor is waiting. A is used to store $\alpha(s_{k_f})$, and t is used to store s_{k_1} (see Definition 4.3). The monitor starts with an initial condition that W is $\mu(s_0)$, and then goes into waiting state, in which it updates W by removing fairness constraints that have been satisfied. Once $W = \emptyset$, a fair prefix is obtained, hence the monitor non-deterministically decides whether k_f is reached, in which case it freezes the abstraction by setting A to $\alpha(s_{k_f})$. It then non-deterministically chooses when to fix k_1. When this happens, the monitor saves s_{k_1} in t, and also resets W to $\mu(s_{k_1})$. It then keeps updating W, and checks if we reach a point in which $W = \emptyset$ and $A(t) = A(s)$, i.e., a point k_2 s.t. $[k_1,k_2]$ is μ-fair, and $\alpha(s_{k_f})(s_{k_1}) = \alpha(s_{k_f})(s_{k_2})$. If this happens, the monitor goes to the error state. The error state is thus reachable in the product of the monitor with the transition system if and only if the (α,μ)-acyclicity condition is violated.

transition system of the ticket protocol. The result is a fair transition system $(\Sigma,\varphi_0,\tau,\Phi)$ defined below, whose fair (infinite) traces represent traces of the protocol that violate the specification, hence its fair termination implies correctness of the protocol:

$$\Sigma = \Sigma^t \cup \{t_0 : \text{thread},\ q\} \qquad \varphi_0 = \varphi_0^t \wedge \neg q$$
$$\tau = \exists x : \text{thread}.\ \tilde{\tau}(x) \wedge t_0' = t_0 \wedge \tau_q \quad \text{where } \tau_q = (\neg q \wedge \neg q') \vee (\neg q \wedge q' \wedge pc_2'(t_0) \wedge \neg pc_3'(t_0)) \vee (q \wedge q' \wedge \neg pc_3'(t_0))$$
$$\Phi = \{\phi_1(x), \phi_2\} \text{ where } \phi_1(x) = \mathit{scheduled}(x),\ \phi_2 = q$$

4 REDUCING FAIR TERMINATION TO SAFETY IN FIRST-ORDER LOGIC

In this section we introduce our approach for proving fair termination using first-order logic. The approach is based on a reduction from fair termination to safety that is applied to transition systems in first-order logic. We start by presenting the reduction and establishing its soundness. Next, we show how to apply the reduction using first-order logic.

4.1 Parametric Reduction

For clarity of the presentation, we first present the reduction of fair termination to safety in a semantic way, ignoring the first-order logic aspects. This semantic reduction is parameterized by a dynamic finite abstraction function, and a dynamic fairness selection function (defined below). In Section 4.2, we show how to algorithmically derive these functions for transition systems that are specified in first-order logic.

Fix a fair transition system $(S,S_0,R,\mathcal{F})$. Roughly speaking, the termination argument for proving that $(S,S_0,R,\mathcal{F})$ has no fair traces is based on showing that no fair abstract cycle exists. To ensure soundness of this argument, we use a dynamic finite abstraction function that must abstract states into a finite set, and we use a dynamic fairness selection function that must select a finite set of fairness constraints, to ensure that eventually they are all satisfied in a finite trace (facilitating the existence of a fair cycle). Formally:

Definition 4.1 (Dynamic Finite Abstraction). A *dynamic finite abstraction function* is a function $\alpha : S \to S \to X$, where X is any set, and for any $s \in S$ the range of $\alpha(s)$ is finite.

Definition 4.2 (Dynamic Fairness Selection). A *dynamic fairness selection function* is a function $\mu : S \to \mathcal{P}_{fin}(\mathcal{F})$ (where $\mathcal{P}_{fin}(\mathcal{F})$ denotes the set of finite subsets of $\mathcal{F}$). Given a trace $(s_i)_{i=0}^{\infty}$ we say that the $[k, k']$ segment of the trace is μ-fair if $\forall F \in \mu(s_k).\ \{s_{k+1}, \ldots, s_{k'-1}\} \cap F \neq \emptyset$.

Recall that we consider transition systems with infinitely many fairness constrains. (In first-order logic transition systems, the fairness formulas induce infinitely many fairness constraints, by instantiating the free variables.) In this setting, an infinite trace can be fair without containing any finite segment that visits all fairness constraints. The dynamic fairness selection function is therefore used to select a finite subset of the fairness constraints that is "relevant" for a particular state (note that each fairness constraint that is selected for $\mu(s)$ is taken as a whole from $\mathcal{F}$; that is, μ does not change the individual fairness constraints, and only selects finitely many fairness constraints for each state). Note that in a fair trace, for every k there exists a $k' > k$ such that the $[k, k']$ segment is μ-fair.

To prove that $(S, S_0, R, \mathcal{F})$ has no fair traces, we require a dynamic finite abstraction function α and a dynamic fairness selection function μ such that the following condition holds:

Definition 4.3 (Acyclicity Condition). The (α, μ)-*acyclicity condition* requires that for any trace $s_0, s_1, \ldots$, for every $k_f < k_1 < k_2$ such that the segments $[0, k_f]$ and $[k_1, k_2]$ are μ-fair, we have $\alpha(s_{k_f})(s_{k_1}) \neq \alpha(s_{k_f})(s_{k_2})$.

Intuitively, the acyclicity condition requires that any fair segment $[k_1, k_2]$ that follows a fair prefix $[0, k_f]$ cannot form a cycle when states are abstracted by the finite abstraction associated with s_{k_f}. The index k_f can therefore be viewed as the "freeze" point of the finite abstraction, after which no fair abstract cycle is allowed. Note that the condition requires that no matter which freeze point is selected (as long as the prefix $[0, k_f]$ is μ-fair), the finite abstraction $\alpha(s_{k_f})$ is precise enough to exclude a fair abstract cycle.

Lemma 4.4. *Let $(S, S_0, R, \mathcal{F})$ and α and μ be s.t. the above conditions are satisfied (Definitions 4.1 to 4.3), then $(S, S_0, R, \mathcal{F})$ has no fair traces.*

Proof. Assume to the contrary that $(s_k)_{k=0}^{\infty}$ is a fair trace of $(S, S_0, R, \mathcal{F})$. Let k_f be an index such that the $[0, k_f]$ segment is μ-fair. Consider the sequence $\left(\alpha(s_{k_f})(s_k)\right)_{k=0}^{\infty}$. Since the range of $\alpha(s_{k_f})$ is finite, there must be an infinite subsequence that consists of a constant $x \in X$. Let $k_1 \geq k_f$ be the index of the first occurrence of x after k_f. Due to fairness, there must exist $k_2 > k_1$ such that the $[k_1, k_2]$ segment is μ-fair. The reason is that a fair trace must visit every element of $\mathcal{F}$ infinitely often. This ensures that whenever we fix k_1 and select the finite subset of $\mathcal{F}$ given by $\mu(s_{k_1})$, we will encounter all elements of $\mu(s_{k_1})$ after some finite number of transitions. In particular, this segment can be extended such that $\alpha(s_{k_f})(s_{k_2}) = x$ (since x repeats infinitely often), i.e., $\alpha(s_{k_f})(s_{k_1}) = \alpha(s_{k_f})(s_{k_2})$. This contradicts the acyclicity condition. □

The acyclicity condition as a safety property. The (α, μ)-acyclicity condition (Definition 4.3) defines a safety property of $(S, S_0, R, \mathcal{F})$. Hence, given α and μ, Lemma 4.4 lets us reduce fair termination to safety checking.

To see this, we present in Fig. 5 a monitor that is parameterized by α and μ. The monitor runs in parallel to the transition system and tracks violations of the (α, μ)-acyclicity condition. Such a violation is a trace $s_0, s_1, \ldots$ of $(S, S_0, R, \mathcal{F})$ where for some k_f the segments $[0, k_f]$ and $[k_1, k_2]$ are μ-fair but $\alpha(s_{k_f})(s_{k_1}) = \alpha(s_{k_f})(s_{k_2})$. To identify the μ-fair prefix $[0, k_f]$, the monitor starts in state "waiting" and uses a set $W \in \mathcal{P}_{fin}(\mathcal{F})$ of fairness constraints that is initialized to $\mu(s_0)$; while the monitor is in state "waiting", W is updated whenever the current state s of the monitored transition system visits one of the fairness constraints. When $W = \emptyset$, the prefix is μ-fair, hence the monitor non-deterministically selects the "freeze" point k_f and stores the corresponding finite abstraction $\alpha(s_{k_f})$ in $A : S \to X$. It then moves to state "frozen" from which it non-deterministically selects k_1, where it saves s_{k_1} in $t \in S$ and initializes W to $\mu(s_{k_1})$ in order to identify a μ-fair segment. It then continues to state "saved" where it updates W in each step, and when $W = \emptyset$, i.e., $[k_1, k_2]$ is a μ-fair segment, if a state s is encountered where $A(s) = A(t)$, then the monitor goes to state "error" and declares violation.

Clearly, the error state is reachable in the product of the monitor with the transition system if and only if the (α, μ)-acyclicity condition is violated. Therefore, the reduction from fair termination to safety constructs the product of the monitor with the transition system. Fair termination then reduces to checking that the product transition system, denoted $(\widehat{S}, \widehat{S_0}, \widehat{R})$, satisfies the safety property $P_{\neg\text{error}}$ defined by the set of states where the monitor is not in its error state. Note that the product transition system has no fairness constraints, since we are only interested in its safety (i.e., in its finite traces). We also note that $(\widehat{S}, \widehat{S_0}, \widehat{R})$ is not an abstract transition system. The use of the dynamic finite abstraction α does not incur any abstraction of the transitions, and it is only used for checking the (α, μ)-acyclicity condition.

Lemma 4.5. *Let $(S, S_0, R, \mathcal{F})$ be a transition system, α a dynamic finite abstraction function and μ a dynamic fairness selection function, and let $(\widehat{S}, \widehat{S_0}, \widehat{R})$ be the transition system obtained from the composition of (S, S_0, R) with the (α, μ)-acyclicity monitor specified in Fig. 6. If $(\widehat{S}, \widehat{S_0}, \widehat{R})$ satisfies the safety property $P_{\neg error}$, then $(S, S_0, R, \mathcal{F})$ has no fair traces.*

4.2 Uniform Reduction in First-Order Logic

We now present the realization of the reduction for a first-order transition system $(\Sigma, \varphi_0, \tau, \Phi)$. The main ingredients are the algorithmic extraction of a dynamic finite abstraction function α and a fairness selection function μ based on the *footprint* of a trace, and the realization of the acyclicity monitor based on these functions as a transition system in first-order logic.

Tracking the footprint of a trace. The dynamic finite abstraction and fairness selection functions formalized in the previous section rely on finiteness arguments. The first key idea that allows us to realize these functions in first-order logic is to augment the first-order transition system $(\Sigma, \varphi_0, \tau, \Phi)$ with a new unary relation d, that will always contain a finite number of elements, and will intuitively accumulate all the elements that the trace has seen or affected so far.

Assume, without loss of generality, that φ_0 and τ are given in the following form:

$$\varphi_0 = \exists x_1, \ldots, x_n.\ \widetilde{\varphi_0}(x_1, \ldots, x_n) \qquad\qquad \tau = \exists x_1, \ldots, x_m.\ \tilde{\tau}(x_1, \ldots, x_m)$$

Then, define an augmented transition system $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$ where

$$\begin{aligned}
\Sigma^d &= \Sigma \cup \{d^1\} \\
\varphi_0^d &= \exists x_1, \ldots, x_n.\ \widetilde{\varphi_0} \wedge \forall x.\ d(x) \leftrightarrow \Big(\textstyle\bigvee_{i=1}^n x = x_i \vee \bigvee_{c \in \Sigma} x = c\Big) \\
\tau^d &= \exists x_1, \ldots, x_m.\ \tilde{\tau} \wedge \forall x.\ d'(x) \leftrightarrow \Big(d(x) \vee \textstyle\bigvee_{i=1}^m x = x_i \vee \bigvee_{c \in \Sigma} x = c'\Big) \\
\Phi^d &= \Phi
\end{aligned}$$

Intuitively, the d relation contains the elements that affect or are affected by the transition, captured by the existentially quantified variables and constants. For both sequential programs and distributed algorithms, each transition usually interacts with a finite set of elements (e.g., threads, memory locations, values, etc.). This set is sometimes called the *footprint* of a transition. The idea of d is that it is updated so that it includes the footprint of all the transitions in the trace so far.

The augmentation of the transition system with d has two benefits, captured by the following lemmas. First, it does not affect termination:

Lemma 4.6. *$(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$ terminates if and only if $(\Sigma, \varphi_0, \tau, \Phi)$ terminates.*

Second, it provides a way to select a finite set of elements in each reachable state:

Lemma 4.7. *Let $s = (\mathcal{D}, \mathcal{I})$ be a reachable state of $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$. Then the interpretation of d, $\mathcal{I}(d) \subseteq \mathcal{D}$, is a finite set.*

The reason is that d initially contains at most $n + |C|$ elements, and with each transition it grows by at most $m + |C|$ elements, where C denotes the set of constant symbols in Σ.

Dynamic finite abstraction and fairness selection functions based on footprints. Now, we apply the reduction from fair termination to safety on $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$. To do so, we define the finite abstraction function for $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$ by projecting all relations to the finite set given by d (at k_f), and we define the fairness selection function by taking all the fairness constraints over all elements of d. Given an interpretation $\mathcal{I}$ and a set A, define the projection of $\mathcal{I}$ to A as $\mathcal{I}|_A = \lambda r.\, \mathcal{I}(r) \cap A^{arity(r)}$. Observe that if A is finite, then the range of $\lambda \mathcal{I}.\mathcal{I}|_A$ is finite. For $s = (\mathcal{D}, \mathcal{I})$, define $s|_A = (A, \mathcal{I}|_A)$.

Definition 4.8 (First-order α and μ). For any state $s = (\mathcal{D}, \mathcal{I})$ of $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$, let

$$\alpha(s) = \lambda s'.\, s'|_{\mathcal{I}(d)}$$

$$\mu(s) = \left\{ F_\phi(\bar{e}) \mid \phi(x_1, \ldots, x_n) \in \Phi,\ \bar{e} \in (\mathcal{I}(d))^n \right\} \quad \text{see Section 3.2 for the definition of } F_\phi(\bar{e})$$

Since $\mathcal{I}(d)$ is finite, the range of $\alpha(s)$ is finite, and so is $\mu(s)$[3].

Implementing the acyclicity monitor in first-order logic. Having defined the dynamic finite abstraction function α and the fairness selection function μ, in order to complete the reduction of fair termination to safety it remains to implement the monitor presented in Fig. 5 for α and μ in first-order logic, and compose it with $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$.

Fig. 6 presents the realization of the monitor in first-order logic, using a vocabulary $\Sigma^m \supseteq \Sigma^d$ which we describe next. The control states of the monitor are tracked using nullary relations *waiting*, *frozen*, *saved*, *error* $\in \Sigma^m$. The key ideas are to "freeze" the abstraction at k_f by copying the relation $d \in \Sigma^d$ at the freeze point into an auxiliary unary relation $a \in \Sigma^m$, and to "select" the relevant subset of the fairness constraints (both initially and when moving to the "saved" state) by introducing an auxiliary relation $w_i \in \Sigma^m$ for each quantified fairness constraint ϕ_i and initializing it to all tuples in d. To implement the check whether $A(t) = A(s)$ (indicating that an abstract state is repeated), the monitor remembers t by introducing a copy $\Sigma_s = \{\ell_s \mid \ell \in \Sigma\}$ of Σ which is set when moving to the "saved" state. Then, the equality of the abstract states is checked by comparing the relation and constant symbols in Σ_s (representing the old state t) and their counterparts in Σ (representing the current state s) on the elements in a. This mimics applying the projection which defines the abstraction at k_f. We denote the first-order specification of the monitor by $(\Sigma^m, \varphi_0^m, \tau^m)$.

The product of $(\Sigma^d, \varphi_0^d, \tau^d, \Phi^d)$ with the monitor is then the first-order transition system $(\widehat{\Sigma}, \widehat{\varphi_0}, \widehat{\tau})$ (with no fairness constraints), where

$$\widehat{\Sigma} = \Sigma^m = \Sigma^d \cup \Sigma_s \cup \{\mathit{waiting}, \mathit{frozen}, \mathit{saved}, \mathit{error}, a\} \cup \{w_i \mid \phi_i \in \Phi\}$$

$$\widehat{\varphi_0} = \varphi_0^d \wedge \varphi_0^m \qquad \text{and} \qquad \widehat{\tau} = \tau^d \wedge \tau^m$$

[3] Strictly speaking, this is only true for the reachable states of $(\Sigma^d, \varphi_0^d, \tau^d)$, since there are unreachable states in which d contains infinitely many elements. However, the soundness result is unaffected by this, since the proof of Lemma 4.4 only applies α and μ to reachable states.

cmd	first-order realization
$W := \mu(s)$	$\bigwedge_i \forall \bar{x}.\ w'_i(\bar{x}) \leftrightarrow \bigwedge_j d(x_j)$
$W := W \setminus \{F \in \mathcal{F} \mid s \in F\}$	$\bigwedge_i \forall \bar{x}.\ w'_i(\bar{x}) \leftrightarrow (w_i(\bar{x}) \land \neg\phi_i(\bar{x}))$
$[W = \emptyset]$	$\bigwedge_i \forall \bar{x}.\ \neg w_i(\bar{x})$
$A := \alpha(s)$	$\forall x.\ a'(x) \leftrightarrow d(x)$
$t := s$	$\bigwedge_{r\in\Sigma} \forall \bar{x}.\ r'_s(\bar{x}) \leftrightarrow r(\bar{x}) \ \land\ \bigwedge_{c\in\Sigma} c'_s = c$
$[A(t) = A(s)]$	$\bigwedge_{r\in\Sigma} \forall \bar{x}.\ \left(\bigwedge_j a(x_j)\right) \to (r_s(\bar{x}) \leftrightarrow r(\bar{x})) \land \bigwedge_{c\in\Sigma}(a(c) \lor a(c_s)) \to c = c_s$

Fig. 6. Realization in first-order logic of commands from Fig. 5. The first-order logic realization uses the following relations: w_i for each $\phi_i \in \Phi$ (w_i has the same arity as ϕ_i), to implement W; a unary relation a, that captures A by recording the interpretation of d at k_f; a relation r_s for every $r \in \Sigma$ (with the same arity), to implement t and capture a copy of the interpretation of all state relations at k_1.

The following theorem summarizes the reduction from fair termination to safety in first-order logic, as well as its correctness:

Theorem 4.9. *Let* $(\Sigma, \varphi_0, \tau, \Phi)$ *be a transition system in first-order logic, and let* $(\widehat{\Sigma}, \widehat{\varphi_0}, \widehat{\tau})$ *be the transition system defined above. If* $(\widehat{\Sigma}, \widehat{\varphi_0}, \widehat{\tau})$ *satisfies the safety property* $\neg error$*, then* $(\Sigma, \varphi_0, \tau, \Phi)$ *has no fair traces.*

The reduction from fair termination to safety is linear (both in time and space) in $(\Sigma, \varphi_0, \tau, \Phi)$. The size of (the description of) the resulting first-order transition system $(\widehat{\Sigma}, \widehat{\varphi_0}, \widehat{\tau})$ is approximately twice as big as that of the input, $(\Sigma, \varphi_0, \tau, \Phi)$ (due to the copied relations).

Handling finite domains. In some settings, we have additional knowledge that some sets of elements are always finite. For example, we could be interested in a distributed or multithreaded protocol without unbounded-parallelism. That is, we consider the protocol only when it is running on a fixed, finite (albeit unbounded) set of nodes or threads. Such a protocol can still use some infinite sorts representing integer values, messages, etc. In such a setting, we may include all threads, nodes, or any other set that is known to be finite into the initial condition for d. Another example is where we might include all elements smaller than some constant representing a natural number initially in d. The soundness of the reduction is maintained, as long as d is guaranteed to be finite in all reachable states.

In particular, in the case of parameterized systems, where all the sorts are finite, we can include *all* elements of the domain in the initial condition of d, while preserving soundness of the reduction. In fact, in this case, fair cycle detection is sound without any abstraction nor fairness selection. Thus, the transition system of the monitor can be simplified by eliminating both d and a, and starting the monitor in the "frozen" state, while initializing W to contain all fairness constraints, and including all elements in the cycle detection (instead of just the ones in a).

Using derived relations and ghost code to increase precision. Our dynamic abstraction of a state is computed completely automatically by projecting the first-order structure to the set of elements determined by the footprint $\mathcal{I}(d)$ at the freeze point. The precision of the abstraction can thus be increased by using additional derived relations, which can allow the projection to distinguish between more states. For example, consider a vocabulary with a unary relation r. Suppose that two states, s_1 and s_2, differ in their interpretation of r, where $\mathcal{I}_1(r) = \emptyset$ and $\mathcal{I}_2(r) = \{e\}$, but e is not included in the finite set by which the abstraction is computed. In this case, $\alpha(s_f)(s_1) = s_1|_{\mathcal{I}_f(d)} = s_2|_{\mathcal{I}_f(d)} = \alpha(s_f)(s_2)$. However, if we add to the vocabulary a nullary derived relation r' that tracks the formula $\exists x.\ r(x)$, then we will now have that $\alpha(s_f)(s_1) \neq \alpha(s_f)(s_2)$, due to the different interpretation of r'. We used such derived relations for the liveness verification of Paxos protocols (see Section 7.1). Another possible way to increase the precision of the abstraction is by adding ghost code that increases the footprint of transitions. This will also cause the finite set of elements to which we project to be larger, making the abstraction more precise.

4.3 Detailed Illustration for Ticket Protocol

We now illustrate our reduction of fair termination to safety on the ticket protocol. Let $(\Sigma, \varphi_0, \tau, \Phi)$ be the fair transition system defined in Example 3.1 for the ticket protocol. To prove the fair termination of $(\Sigma, \varphi_0, \tau, \Phi)$ we apply our reduction to safety. The first step is to augment the transition system by tracking the footprint. This results in the following transition system, T^d (note that we add a unary relation for each sort):

$$
\begin{aligned}
\Sigma^d &= \Sigma \cup \{d_{\text{thread}}(\text{thread}),\ d_{\text{ticket}}(\text{ticket})\}\\
\varphi_0^d &= \varphi_0 \land (\forall x : \text{thread}.\ d_{\text{thread}}(x) \leftrightarrow (x = t_0)) \land (\forall x : \text{ticket}.\ d_{\text{ticket}}(x) \leftrightarrow (x = n \lor x = s))\\
\tau^d &= \exists x : \text{thread}.\ \tilde{\tau}(x) \land t_0' = t_0 \land \tau_q \land \big(\forall y : \text{thread}.\ d'_{\text{thread}}(y) \leftrightarrow (d_{\text{thread}}(y) \lor x = y)\big) \land\\
&\qquad \big(\forall y : \text{ticket}.\ d'_{\text{ticket}}(y) \leftrightarrow (d_{\text{ticket}}(y) \lor y = n' \lor y = s')\big)
\end{aligned}
$$

Next, we construct the first-order realization of the monitor of Fig. 5 and compose it with the augmented transition system. This completes the reduction from fair termination to safety reduction, and results in the following transition system, $\hat{T}$:

$$
\begin{aligned}
\widehat{\Sigma} &= \Sigma^d \cup \{r_s \mid r \in \Sigma\} \cup \{\mathit{waiting},\ \mathit{frozen},\ \mathit{saved},\ \mathit{error}\}\\
&\qquad \cup \{a_{\text{thread}}(\text{thread}),\ a_{\text{ticket}}(\text{ticket}),\ w_1(\text{thread}),\ w_2\}\\
\widehat{\varphi_0} &= \varphi_0^d \land \mathit{waiting} \land \neg \mathit{frozen} \land \neg \mathit{saved} \land \neg \mathit{error} \land (\forall x : \text{thread}.\ w_1(x) \leftrightarrow d_{\text{thread}}(x)) \land w_2\\
\widehat{\tau} &= \tau^d \land (\tau_{\text{wait}} \lor \tau_{\text{freeze}} \lor \tau_{\text{save}} \lor \tau_{\text{error}})\\
\tau_{\text{wait}} &= (\mathit{waiting}' \leftrightarrow \mathit{waiting}) \land (\mathit{frozen}' \leftrightarrow \mathit{frozen}) \land (\mathit{saved}' \leftrightarrow \mathit{saved}) \land\\
&\qquad \big(\forall x : \text{thread}.\ w_1'(x) \leftrightarrow (w_1(x) \land \neg \mathit{scheduled}(x))\big) \land (w_2' \leftrightarrow (w_2 \land \neg q))\\
\tau_{\text{freeze}} &= \mathit{waiting} \land \neg \mathit{waiting}' \land \mathit{frozen}' \land \neg \mathit{saved}' \land (\forall x : \text{thread}.\ \neg w_1(x)) \land \neg w_2 \land\\
&\qquad \big(\forall x : \text{thread}.\ a'_{\text{thread}}(x) \leftrightarrow d_{\text{thread}}(x)\big) \land \big(\forall x : \text{ticket}.\ a'_{\text{ticket}}(x) \leftrightarrow d_{\text{ticket}}(x)\big)\\
\tau_{\text{save}} &= \mathit{frozen} \land \neg \mathit{waiting}' \land \neg \mathit{frozen}' \land \mathit{saved}' \land \big(\forall x : \text{thread}.\ w_1'(x) \leftrightarrow d_{\text{thread}}(x)\big) \land w_2' \land\\
&\qquad n_s' = n \land s_s' = s \land q_s' \leftrightarrow q \land \forall x : \text{thread}.\ m_s'(x) = m(x) \land \textstyle\bigwedge_k {pc_k}_s'(x) \leftrightarrow pc_k(x)\\
\tau_{\text{error}} &= \mathit{saved} \land \mathit{error}' \land (\forall x : \text{thread}.\ \neg w_1(x)) \land \neg w_2 \land q_s \leftrightarrow q \land\\
&\qquad \big(\forall x : \text{thread}.\ a_{\text{thread}}(x) \to \textstyle\bigwedge_k {pc_k}_s(x) \leftrightarrow pc_k(x)\big) \land\\
&\qquad (\forall x : \text{ticket}.\ a_{\text{ticket}}(x) \to (n_s = x \leftrightarrow n = x) \land (s_s = x \leftrightarrow s = x)) \land\\
&\qquad (\forall x : \text{thread}, y : \text{ticket}.\ a_{\text{thread}}(x) \land a_{\text{ticket}}(y) \to (m_s(x) = y \leftrightarrow m(x) = y))
\end{aligned}
$$

Inductive invariant. To provide a better understanding of the entire verification process, we discuss the inductive invariant that establishes the safety of the system $\hat{T}$ that results from the reduction for the example of the ticket protocol. The inductive invariant includes some rather straightforward properties of reachable states of the ticket protocol. The most interesting part of the inductive invariant is the one that establishes the connection between the protocol state and the footprint, as well as the absence of fair abstract cycles. This part is:

$$\begin{aligned}
&\forall x : \text{thread}.\ (pc_2(x) \lor pc_3(x)) \to d_{\text{thread}}(x)\\
&\forall x : \text{ticket}.\ x \le n \to d_{\text{ticket}}(x)\\
&(\mathit{frozen} \lor \mathit{saved}) \to \forall x : \text{thread}.\ (pc_2(x) \lor pc_3(x)) \land m(x) \le m(t_0) \to a_{\text{thread}}(x)\\
&(\mathit{frozen} \lor \mathit{saved}) \to \forall x : \text{ticket}.\ x \le m(t_0) \to a_{\text{ticket}}(x)\\
&\forall x : \text{ticket}.\ s \le x < n \to \exists y : \text{thread}.\ m(y) = x \land (pc_2(y) \lor pc_3(y))\\
&\mathit{saved} \to \forall x : \text{thread}.\ \big(m_s(x) = s_s \land pc_{2_s}(x) \land \lnot w_1(x)\big) \to\\
&\qquad\qquad\big((pc_1(x) \land m(x) = s_s) \lor (pc_2(x) \land m(x) > m(t_0)) \lor (pc_3(x) \land m(x) = s_s)\big)\\
&\mathit{saved} \to \forall x : \text{thread}.\ \big(m_s(x) = s_s \land pc_{3_s}(x) \land \lnot w_1(x)\big) \to\\
&\qquad\qquad((pc_1(x) \land m(x) = s_s) \lor (pc_2(x) \land m(x) > m(t_0)))
\end{aligned}$$

Note that this invariant establishes the fact that all threads in pc_2 or pc_3 are in d_{thread}, and that all allocated ticket numbers are in d_{ticket}. It then establishes that after the freeze point, a_{thread} and a_{ticket} include all the threads ahead of t_0, and all the ticket numbers lower than the ticket of t_0. The rest of the invariant establishes the existence of the thread whose state must change in every fair segment. Notice in particular the use of $\lnot w_1(x)$, which expresses the fact that thread x has been scheduled since the save point (for threads that were active at the save point). This structure is typical, and represents the structure of the inductive invariants we obtained in all of our examples.

We note that for the ticket example, the resulting verification conditions fall into the decidable EPR fragment of first-order logic (all quantifier-alternations are stratified). This means that for the ticket protocol, our reduction actually allows decidable deductive verification of liveness.

5 CAPTURING NESTED TERMINATION ARGUMENTS

While sound, the reduction presented in the previous section loses completeness even for systems that are of practical interest. In particular, it does not capture a form of termination arguments we call *nested*, that are needed for interesting protocols. In this section, we present a more powerful reduction that relies on a user-provided *nesting structure*, that captures such arguments. We start by presenting the example of the alternating bit protocol for motivation and intuition. We show that for this protocol, the reduction presented in Section 4 is incomplete, i.e., it results in an unsafe transition system. We then present the more powerful reduction based on a nesting structure.

5.1 Alternating Bit Protocol

The alternating bit protocol (ABP) is a classic communication algorithm for transition of messages using lossy first-in-first-out (FIFO) channels. The protocol involves a sender and a receiver, which operate according to the code in Fig. 7. The protocol uses two channels, the `data` channel from the sender to the receiver, and the `ack` channel, from the receiver to the sender. The sender and the receiver each have a state bit, initialized to 0, which acts as a "sequence number". We assume that the sender initially has an (infinite) array `sender_array` filled with values that we wish to transmit to the receiver. The sender keeps sending the first value with a "sequence number bit" of 0. When the receiver receives this value, it will store it in `receiver_array`, flip its bit, and start sending an acknowledgment message for sequence bit 0. When the sender receives this, it will also flip its bit and start sending the second value from the array with sequence bit 1. The sender and receiver realize when they should move on to the next value by comparing their bit to the

```
initially: sender_i = receiver_i = 0, sender_bit = receiver_bit = 0
```

```
process sender

action send_data:
    data_channel.send(
        sender_array[sender_i], sender_bit)

action receive_ack:
    b := ack_channel.receive()
    if b = sender_bit:
        sender_i := sender_i + 1
        sender_bit := !sender_bit
```

```
process receiver

action receive_data:
  d, b := data_channel.receive()
  if b = receiver_bit:
      receiver_array[receiver_i] := d
      receiver_i := receiver_i + 1
      receiver_bit := !receiver_bit

action send_ack:
    if receiver_i > 0:
        ack_channel.send(!receiver_bit)
```

Fig. 7. The alternating bit protocol for transition of messages using lossy first-in-first-out (FIFO) channels. The protocol comprises of two processes, `sender` and `receiver`, and two lossy FIFO channels, `data` and `ack`.

sequence bit of the incoming messages. Assuming the array is infinite, the protocol continues to send values and the receiver keeps receiving them.

For this protocol, we wish to prove the following progress property, that states that every element of the sender's data array is eventually transferred to the receiver:

$$\forall i.\ \Diamond\ \mathit{receiver\_array}[i] = \mathit{sender\_array}[i]$$

The protocol satisfies the above property under suitable fairness assumptions. For simplicity of the presentation, in this section we take the fairness assumptions to be that the 4 actions `send_data`, `send_ack`, `receive_ack`, and `receive_data` are called infinitely often. The more realistic fairness assumptions are explained in Section 7.

As explained in Section 3.4, verifying the progress property under the fairness assumptions is equivalent to proving termination of the fair transition system obtained by the product of the system with a "monitor" for the (Skolemized) negation of the specification:

$$\left(\bigwedge_{A\in\mathcal{A}} \Box\Diamond\, A \text{ is called}\right) \wedge \Box\ \mathit{receiver\_array}[i_0] \neq \mathit{sender\_array}[i_0]$$

where $\mathcal{A} = \{$`send_data`, `send_ack`, `receive_ack`, `receive_data`$\}$.

5.2 Inadequacy of the Fair Termination to Safety Reduction for ABP

The ABP protocol satisfies its specification, which ensures that the above fair transition system has no fair traces. Nevertheless, applying the liveness-to-safety reduction of Section 4 results in an unsafe transition system. This is already the case for the parametric reduction described in Section 4.1 with any α and μ, due to the following lemma:

Lemma 5.1. *Let $(S^{ABP}, S_0^{ABP}, R^{ABP}, \mathcal{F}^{ABP})$ be the fair transition system whose fair termination characterizes the correctness of ABP. The (α,μ)-acyclicity condition does not hold for any dynamic finite abstraction function α and fairness selection function μ.*

Proof. First, we observe that the (α,μ)-acyclicity condition is monotone in μ. Namely, the more fairness constraints μ selects, the "easier" it is for the acyclicity condition to hold. Since in this case $\mathcal{F}^{\text{ABP}}$ is finite (it contains 4 constraints, corresponding to the four actions in $\mathcal{A}$), we restrict ourselves to the case of $\mu = \lambda s.\ \mathcal{F}^{\text{ABP}}$, and show that no suitable α exists. (As explained above, if the condition holds for some α and μ, it will also hold with this μ.)

We prove the lemma by providing a family of execution prefixes of the form $w_f \cdot w_n$ for $n \in \mathbb{N}$, such that w_f is fair (i.e., visits all fairness constraints), and w_n contains n fair segments (i.e., n

disjoint segments, each of them visits all the fairness constraints). Now, consider any dynamic finite abstraction function α and assume that it satisfies the acyclicity condition. Let s_f be the state at the end of w_f. The range of $\alpha(s_f)$ must be finite, and let N denote its cardinality. Now, consider the execution prefix $w_f \cdot w_{N+2} = s_0, \ldots, s_m$. Let $k_f < k_1 < \ldots < k_{N+1}$ be such that k_f is the index of the last state in w_f (so $[0, k_f]$ is fair, and $s_{k_f} = s_f$), and such that for every $1 \leq i \leq N$, the segment $[k_i, k_{i+1}]$ is fair. Now, by the acyclicity condition $\alpha(s_f)(s_{k_i}) \neq \alpha(s_f)(s_{k_j})$ for every $1 \leq i < j \leq N+1$. However, the cardinality of the range of $\alpha(s_f)$ is N, so by the pigeonhole principle we reached a contradiction.

To see that the alternating bit protocol contains such a family of execution prefixes, take:

$$\begin{aligned}
w_f &= (\mathtt{send\_data}) \cdot (\mathtt{receive\_data}) \cdot (\mathtt{send\_ack}) \cdot (\mathtt{receive\_ack}) \\
w_n &= y_n \cdot z_n \text{ where:} \\
y_n &= (\mathtt{send\_data})^n \cdot (\mathtt{send\_ack})^n \cdot (\mathtt{receive\_data}) \cdot (\mathtt{receive\_ack}) \\
z_n &= ((\mathtt{send\_data}) \cdot (\mathtt{receive\_data}) \cdot (\mathtt{send\_ack}) \cdot (\mathtt{receive\_ack}))^{n-1}
\end{aligned}$$

First, note that w_f is fair, y_n is fair, and z_n contains $n-1$ fair segments, so w_n indeed contains n fair segments. Let us consider the values transferred in such an execution prefix. After w_f, the first value of the array is transferred to the receiver. After y_n, the second value is transferred, but the data channel contains $n-1$ copies of the second value, and the acknowledgment channel contains $n-1$ acknowledgments for the first value. After z_n, these $n-1$ copies are all received, but the third value of the sender array has not yet been transferred to the receiver. Therefore, if we take $i_0 = 2$ (the index of the third value), then these are indeed execution prefixes of $(S^{\mathrm{ABP}}, S_0^{\mathrm{ABP}}, R^{\mathrm{ABP}}, \mathcal{F}^{\mathrm{ABP}})$. $\square$

The key idea of the above proof is that after the freeze point k_f, any given α actually bounds the number of fair segments that the system can take. However, as we saw, for the alternating bit protocol there can be no such bound: there is no bound on the number of (fair) steps that are needed from the time a message is sent for index i to the time its acknowledgment is received.

However, once a data message for some value is sent, we *can* give a bound on how many fair segments can happen before it is received — the bound is the amount of messages ahead of it in the channel (note that even if it is dropped, it will be re-transmitted). A similar argument holds for acknowledgment messages — once an acknowledgment message is sent, we can bound the number of fair segments before it (or a copy of it) is received. Intuitively, this gives the alternating bit protocol a flavor of a nested loop. The outer loop iterates through the array, each iteration of it corresponds to a new value being transmitted. The inner loop keeps retransmitting data and acknowledgment messages until they are received.

This intuition leads to a more general liveness-to-safety reduction, which can prove the liveness of the alternating bit protocol. The motivating idea is to split the transitions of the transition system into several nested levels, and to prove the fair termination of every level separately, while assuming the inner levels terminate. For the alternating bit protocol, the inner level will consist of transitions in which the sender bit and the receiver bit do not change, and the outer level will consist of transitions which change those bits.

Next, we formalize this nested reduction. We later return to the alternating bit protocol and explain its proof.

5.3 Reduction with Nesting Structure

Fix a fair transition system $(S, S_0, R, \mathcal{F})$. In order to support termination proofs of systems such as the alternating bit protocol, we introduce a nesting structure.

Definition 5.2 (Nesting Structure). For a fair transition system $(S, S_0, R, \mathcal{F})$, a *nesting structure* with n levels is $\bar{\eta} = \langle \eta^0, \ldots, \eta^{n-1} \rangle$ such that for every $0 \leq i < n$, $\eta^i \subseteq S \times S$, $\eta^0 = R$, and for every $0 \leq i < n-1$, $\eta^{i+1} \subseteq \eta^i$. Given a nesting structure, and a trace $\pi = s_0, s_1, \ldots$, we define the *level* of position k by $level_\pi(k) = \max\{i \mid (s_k, s_{k+1}) \in \eta^i\}$.

To prove that $(S, S_0, R, \mathcal{F})$ has no fair traces, we require a nesting structure $\bar{\eta}$, a dynamic finite abstraction function α and a dynamic fairness selection function μ such that the following condition holds:

Definition 5.3 (Nested Acyclicity Condition). The $(\bar{\eta}, \alpha, \mu)$-*nested acyclicity condition* requires that for any trace $\pi = s_0, s_1, \ldots$, for any nesting level $0 \leq i < n$ and for any $k_e \leq k_f < k_1 < k_2$ such that: (i) $\forall k_e \leq k < k_2.\ level_\pi(k) \geq i$, and (ii) $level_\pi(k_1) = level_\pi(k_2) = i$, and (iii) the segments $[k_e, k_f]$ and $[k_1, k_2]$ are μ-fair, we have $\alpha(s_{k_f})(s_{k_1}) \neq \alpha(s_{k_f})(s_{k_2})$.

By augmenting a transition system with a nesting structure, we decompose the fair termination proof to a proof for each level, assuming the subsequent levels terminate. This is analogous to proving that a program with nested loops terminates by proving that each loop terminates under the assumption that the inner ones terminate.

Lemma 5.4. *Let $(S, S_0, R, \mathcal{F})$ and $\bar{\eta}$ and α and μ be s.t. the above conditions are satisfied (Definitions 4.1, 4.2, 5.2 and 5.3), then $(S, S_0, R, \mathcal{F})$ has no fair traces.*

Proof. Assume to the contrary that $\pi = (s_k)_{k=0}^{\infty}$ is a fair trace of $(S, S_0, R, \mathcal{F})$. Consider the sequence $(level_\pi(k))_{k=0}^{\infty}$, and let i_m be the minimal level that appears infinitely often in this sequence. We can now choose a subsequence $k_0 < k_1 < \ldots$ s.t. $level_{k_i} = i_m$ and $\forall k_0 \leq k.\ i_m \leq level_k$. Let j_f be the first index such that the $[k_0, k_{j_f}]$ segment is μ-fair. Consider the sequence $\left(\alpha(s_{k_{j_f}})(s_{k_j})\right)_{j=0}^{\infty}$. Since the range of $\alpha(s_{k_{j_f}})$ is finite, there must be an infinite subsequence which is a constant $x \in X$. Let $j_1 \geq j_f$ be the index of the first occurrence of x after j_f. Due to fairness, there must exist $j_2 > j_1$ such that the $[k_{j_1}, k_{j_2}]$ segment is μ-fair. In particular, this segment can be extended such that $\alpha(s_{k_f})(s_{k_{j_2}}) = x$ (since x repeats infinitely often), i.e., $\alpha(s_{k_f})(s_{k_{j_1}}) = \alpha(s_{k_f})(s_{k_{j_2}})$. Now, for $k_e = k_0$, $k_f = k_{j_f}$, $k_1 = k_{j_1}$ and $k_2 = k_{j_2}$, this contradicts the nested acyclicity condition. □

A monitor for the nested acyclicity condition. Similarly to the non-nested acyclicity condition, we depict in Fig. 8 a monitor, parameterized by α, μ and $\bar{\eta}$, that tracks violations of the $(\bar{\eta}, \alpha, \mu)$-acyclicity condition. The key difference compared to the monitor of the non-nested condition is the tracking of the nesting level. To identify violations of the condition in every nesting level, the monitor non-deterministically selects a level $0 \leq i < n$. It then uses the self edge in state "waiting" to non-deterministically select the "entry" point k_e to level i after which an abstract fair cycle in level i will be detected. From this point on, it makes sure that the level remains at least i, and that the states that close an abstract cycle are both encountered at level exactly i. If this happens, a violation is detected and the monitor moves to state "error".

When using the monitor from Fig. 8 instead of the one given in Fig. 6, Lemma 4.5 extends to formalize the soundness of the reduction of fair termination to safety verification for every α, μ, and nesting structure $\bar{\eta}$.

Realization in first-order logic. To realize the reduction using a nesting structure in first-order logic, we let the user specify the nesting structure $\bar{\eta} = \langle \eta^0, \ldots, \eta^{n-1} \rangle$ by providing $n-1$ two vocabulary formulas $\varphi_\eta^1, \ldots, \varphi_\eta^{n-1}$ (i.e., formulas over $\Sigma \cup \Sigma'$), and then define:

$$\eta^i = \left\{ (s, s') \in S \times S \mid (s, s') \models \left(\tau \wedge \bigwedge\nolimits_{1 \leq j \leq i} \varphi_\eta^j \right) \right\}$$

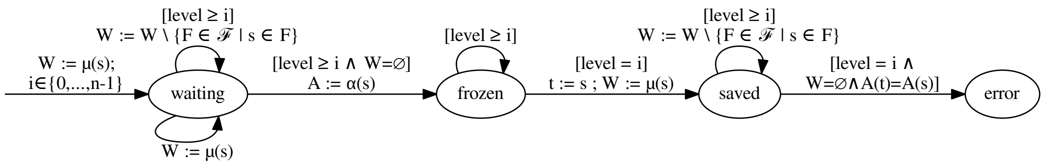


Fig. 8. Monitor that checks the $(\bar{\eta}, \alpha, \mu)$-acyclicity condition. The monitor extends the monitor depicted in Fig. 5 to also track the nesting level. It uses an auxiliary state $0 \leq i < n$, which is initialized non-deterministically. The variable *level* denotes the level of the current position of the monitored transition system, computed based on $\bar{\eta}$. The monitor non-deterministically selects when to fix k_e, the entry position to level i, and from that point on behaves similarly to the monitor in Fig. 5, except that it also makes sure that the level always remains at least i, and that the level of the points k_1 and k_2 where $\alpha(s_{k_f})(s_{k_1}) = \alpha(s_{k_f})(s_{k_2})$ is exactly i. The error state is reachable in the product of the monitor and the monitored transition system if and only if the $(\bar{\eta}, \alpha, \mu)$-acyclicity condition is violated.

Note that this satisfies $\eta^0 = R$, and for every $0 \leq i < n - 1$, $\eta^{i+1} \subseteq \eta^i$. With this definition, it is straightforward to construct a first-order transition system that implements the monitor of Fig. 8. We also note that for the degenerate case of $n = 1$, i.e., a single level, this construction is identical to that of Section 4.

Nesting structure for ABP. Finally, we demonstrate the use of a nesting structure on the alternating bit protocol. For the alternating bit protocol, we need 2 levels, which means we specify $\bar{\eta}$ by providing the following formula: $\varphi^1_\eta = (sender\_bit' \leftrightarrow sender\_bit) \land (receiver\_bit' \leftrightarrow receiver\_bit)$. This lets the "inner loop" consist of the transitions that do not change the sender or the receiver's bit, and effectively lets the proof of the "outer loop" assume that in any fair trace, the bits change infinitely often. Using this nesting structure, the resulting transition system is safe, and we are able to verify its safety using an inductive invariant in first-order logic (actually, in EPR). More interesting details about this example appear in Section 7.

6 LIMITATIONS OF OUR REDUCTION IN FIRST-ORDER LOGIC

Incompleteness of first-order inductive invariant for proving safety. One kind of incompleteness that affects the verification process occurs after our liveness-to-safety reduction. It could be the case that the transition system produced by our reduction is safe, but its safety cannot be proven by any inductive invariant expressible in first-order logic. In such cases, one can employ the known methods of adding ghost code and auxiliary predicates (to the resulting system) to express the inductive invariant. However, even with these methods, there is no complete proof system for safety of first-order transition systems, as verifying their safety is in general undecidable.

For the rest of this section, we discuss incompleteness of the reduction itself, i.e., cases where the resulting transition system is unsafe.

Incompleteness of the reduction in first-order logic. While sound, our reduction is incomplete. As demonstrated in Section 5.2 for the non-nested reduction, some incompleteness is incurred already for the parametric reduction, i.e., for any choice of α and μ. That is, there are some fair transition systems with no infinite fair traces, such that for any α and μ, the reduction results in an unsafe transition system. However, we consider the more interesting incompleteness incurred by our uniform α and μ in first-order logic, that is, the use of the footprint and structure projection. In this

```
            initially: y = 0 ∧ ¬flag                P1(y) := while (y > 0)
                                                                y := y − 1;
while (¬flag)              ||  flag := true;        P2(y) := y := y + 2;
    y := y + 1;            ||                                while (y > 0)
P(y)                       ||                                    y := y − 1;
                           ||                       P3(y) := compute Ackermann(y, y)
```

Fig. 9. Program that non-deterministically chooses a natural number y and performs a computation $P(y)$ depending on y. The different programs P_1,P_2,P_3 are used to demonstrate the limits of our verification technique.

section, we shed some light on this incompleteness using examples in which it manifests. However, we start by a simple example for which the reduction is successful, to illustrate the small differences than can cause it to fail.

The AnyY program. Consider the program in Fig. 9 where $P(y)$ is instantiated with $P_1(y)$. The program contains two threads, and we assume they are scheduled fairly. Consider the transition system that results from applying the reduction of Section 4.2 to this program. In this transition system, the freeze point will be after the flag is set to true (since we wait for both threads to be scheduled). At that point, the footprint will contain all elements seen so far, which are all elements smaller than or equal to y. By projecting to this finite set, the abstraction can distinguish between all states of the remaining execution. Therefore, no abstract fair cycle is present, and the transition system is safe.

The AnyY+2 program. Now, consider the same program in Fig. 9 where $P(y)$ is instantiated with $P_2(y)$. If we use the footprint to define α as before, the resulting abstraction will not be able to distinguish between the values $y_f + 1$ and $y_f + 2$, where y_f is the value of y at the freeze point, since both these values will not be in the relation d. As a result, the obtained transition system will not be safe. This happens despite the fact that with the parametric version of the reduction of Section 4, we can actually find a suitable α such that the resulting transition system is safe (for μ we simply take both fairness constraints, as there are only two). Take $\alpha = \lambda s.\ \lambda s'.\min(s(y) + 2, s'(y))$, where $s(y)$ denotes the value of y at state s. Thus, $\alpha(s)$ abstracts states into the finite set $\{0, \dots, s(y) + 2\}$. This abstraction distinguishes between all states of the remaining execution, so the reduction results in a safe transition system.

While the first-order reduction of Section 4.2 fails for this program, the first-order reduction of Section 5 can succeed, if we use a 2-level nesting structure that puts the while loop in $P_2(y)$ into a separate level.

The AnyAck program. As seen in the proof of Lemma 5.1, for a program with a finite set of fairness constraints, the (α,μ)-acyclicity condition requires that from any reachable freeze point s, the system cannot take more than N fair segments after the freeze point, where N is the cardinality of $\alpha(s)$. In first order logic, $N = O(exp(poly(k)))$, where k is the length of the path from the initial state to the freeze point. This is because the size of the footprint d is linear in k, and the number of different projections of structures from a fixed vocabulary to a set of n elements is $O(exp(poly(n)))$.

Applying this argument to programs of the form of Fig. 9, we obtain that the reduction of Section 4.2 must fail if $P(y)$ takes more than exponential time in y. By using a nesting structure and the reduction of Section 5, this limitation is lifted. However, one can still show that the number of steps that $P(y)$ may do, can be computed by a primitive recursive function over y. This means that if we let $P(y)$ be a program that computes the Ackermann function of y, no nesting structure can make the reduction in first-order logic to work. This is in sharp contrast to the parametric reduction, which can prove programs of this form for any terminating $P(y)$ without a nesting structure at all, simply by letting $\alpha(s)$ contain as many elements as the number of steps $P(s(y))$ takes to terminate.

Summary of limitations. In Section 5.2 we already saw an example where the parametric reduction without nesting function fails, but adding a nesting structure lets even the first-order reduction succeed. In this section, we saw examples for which the parametric reduction succeeds without a nesting structure (for a suitable α), but the first order reduction either requires a nesting structure (AnyY+2), or even fails for any nesting structure (AnyAck). This shows that the theoretical characterization of the proof theoretic power of our formalism is intriguing, and we consider it an attractive direction for future investigation. However, from the practical perspective, when considering distributed protocols (that do not ordinarily lead to non-primitive recursion), we expect our reduction in first-order using a nesting structure to provide a powerful enough proof system. This is already demonstrated by the challenging examples we considered in this work.

7 EVALUATION

To evaluate the applicability of the presented approach, we used it to verify liveness properties of several challenging protocols. For safety verification, we used the IVy tool [McMillan 2016; Padon et al. 2016], which uses the Z3 theorem prover [de Moura and Bjørner 2008] to discharge verification conditions. For each example, we manually applied our reduction to obtain a safety verification problem. We then used IVy to interactively find an inductive invariant that proves safety of the resulting system, and to automatically check the resulting verification conditions. For all examples except the TLB shootdown, the resulting verification conditions are in the EPR decidable fragment. However, even for the TLB shootdown, Z3 was able to verify the inductive invariant in a few minutes. Fig. 10 lists the examples, along with the run times for checking verification conditions. The artifact of this paper [4] contains the IVy files for all described examples.

Next we discuss the interesting features of each example (Section 7.1), and the user experience and effort required in the verification process (Section 7.2), which is an important aspect in evaluating the practicality of our approach.

7.1 Examples

Ticket Protocol. The example of the ticket protocol has been discussed in detail throughout the paper. We note that in order to have the verification conditions in EPR, we modeled the local variable m using a relation, rather than a function. This allows us to use a $\forall$ticket. $\exists$thread quantifier alternation which is needed in the inductive invariant (see Section 4.3), without breaking stratification. We also emphasize that our proof shows the non-starvation even for the case of unbounded-parallelism, i.e., unbounded dynamic thread creation.

Alternating Bit Protocol. The alternating bit protocol was described in Section 5.1. Our model and proof are naturally performed in EPR. We model the FIFO channels using dynamic totally ordered sets. Section 5.1 presented simplified fairness constraints. For the evaluation, we used the standard fairness constraints for each channel: if messages are infinitely often sent, then messages are infinitely often received. The proof is done using a nesting structure with 2 levels, as described in Section 5.

Another interesting feature of this example is that we must assume there are only finitely many indices smaller than i_0 (the Skolem constant from the negation of $\forall i.\ \Diamond\ receiver\_array[i] =$

[4] Available at http://www.cs.tau.ac.il/~odedp/reducing-liveness-to-safety-in-first-order-logic/.

Protocol	$< \infty$	∞	$\lvert\bar{\eta}\rvert$	$\mathbf{C}_s$	$\mathbf{C}_e$	$\mathbf{C}_m$	VC	t [sec]
Ticket Protocol	–	threads, tickets	1	15	3	19	EPR	3.7
Alternating Bit Protocol	$\{i : \text{index} \mid i < i_0\}$	indices, values, messages	2	13	2	20	EPR	6.9
TLB Shootdown	–	processors, pagemaps, entries	3	22	9	60	FO	219
Paxos	nodes	ballots, values	1	9*	11	22	EPR	8.4
Multi-Paxos	nodes	ballots, values, instances	1	10*	13	32	EPR	9.0
Stoppable Paxos	nodes	ballots, values, instances	1	14*	14	34	EPR	9.9

Fig. 10. Protocols for which we verified liveness. For each protocol, $< \infty$ reports what is assumed to be finite (but unbounded), and ∞ reports what is allowed to be truly infinite. $|\bar{\eta}|$ reports the number of nesting levels used in the proof. $\mathbf{C}_s$, $\mathbf{C}_e$, and $\mathbf{C}_m$ list the number of conjectures used in the inductive invariant (which provides some measure of user effort, see Section 7.2), split into: conjectures used to prove safety of the original protocol ($\mathbf{C}_s$), conjectures added for the liveness proof that express additional properties of the original protocol ($\mathbf{C}_e$), and conjectures that prove the safety of the transition system resulting from the liveness-to-safety reduction by relating the state of the protocol and the state of the monitor ($\mathbf{C}_m$). VC mentions if the resulting verification conditions are in EPR or FO (general first-order logic). Finally, **t** reports the run time (in seconds) for checking the verification conditions using IVy and Z3. The experiments were performed on a laptop running 64-bit Linux, with a Core-i7 1.8 GHz CPU. Z3 version 4.5.0 was used, along with the latest version of IVy (commit 7ce6738). Z3 uses heuristics which employ randomness. Therefore, each experiment was repeated 10 times using random seeds, and we report the mean.
* For the Paxos examples, not all conjectures in $\mathbf{C}_s$ were used in the liveness proof, in order to avoid quantifier alternations that would result in verification conditions outside of EPR. Nevertheless, $\mathbf{C}_s$ counts all conjectures needed to prove safety, in order to compare the difficulty of proving safety and the difficulty of proving liveness.

$sender\_array[i]$). Indeed, for a domain in which the indices are a total order which includes unreachable elements, the system does not actually satisfies the progress property for every i_0. The assumption that i_0 is reachable is cleanly expressed in our formalism by letting the initial condition for d include all indices smaller than i_0. A possible alternative to this would be to explicitly model a concurrent action that fills the sender array with values, and rephrase the progress property such that for every array entry, once it is filled at the sender it must eventually be copied to the receiver. This will actually have the same effect, as it will make sure that by the freeze point, all elements smaller than i_0 are included in d.

We note that the alternating bit protocol was also considered by [Abdulla et al. 2006]. The liveness property they verify is that the sender and the receiver change their bits infinitely often. Thus, comparing with our proof, they only verified that the "inner-loop" terminates, while we prove the more natural specification that every array entry is eventually transmitted.

TLB Shootdown. The TLB shootdown algorithm [Black et al. 1989] is used in the Mach operating system. Modern processors use page tables to translate from virtual to physical memory. These page tables are cached in the processors in the Translation Look-aside Buffer (TLB). When some processor changes the page table, it interrupts all other processors currently using the page table and waits for them to receive the interrupt before changing the entries. In [Hoenicke et al. 2017], an abstracted version of this algorithm was formally verified, after adding one critical atomic region to the code that prevents an error path. This only showed safety of this algorithm. To the best of our knowledge, we are the first to mechanically prove its liveness property.

The algorithm itself runs in four phases: (1) the initiator interrupts all processors using a page table, (2) the interrupted processors set a flag that they are deactivated, (3) when every processor set the flag, the initiator changes the page table and finishes, (4) the responders can then continue and flush their TLB. The algorithm is further complicated by the fact that a processor can non-deterministically choose to act as initiator in which case it doesn't respond to interrupts.

We show that each processor will either infinitely often run through the main loop or infinitely often trough the responder loop, which shows that both the initiator and the loop body of the responder terminate. Similarly to the ticket protocol, we implemented the lock operation as spin-locks to expose dead-locks as non-terminating runs. We also added strong fairness assumptions for the page table lock; otherwise, a process can be blocked indefinitely when waiting for the lock.

We took the code from [Hoenicke et al. 2017] and translated it into IVy. The resulting transition system has 24 program locations. To apply our reduction, we use a nesting structure with 3 levels. Interestingly, our proof is sound even in the case where processors are added dynamically and the number of processors is unbounded.

Paxos, Multi-Paxos, and Stoppable Paxos. The Paxos family of protocols is widely used in practice to build fault-tolerant distributed systems, and verifying the safety and liveness of protocols in the family constitutes a current verification challenge [Dragoi et al. 2016; Hawblitzel et al. 2015; Padon et al. 2017; Wilcox et al. 2015]. All protocols in the family are variations on the Paxos consensus algorithm [Lamport 1998, 2001]. Under certain fairness assumptions, Paxos allows a set of crash-prone nodes in an asynchronous network to solve the consensus problem, i.e. to agree on a common decision taken among values that the nodes propose (fairness assumptions are unavoidable, as purely asynchronous fault-tolerant distributed consensus is impossible [Fischer et al. 1985]). We prove that Paxos eventually reaches a decision provided that there is a node l and a majority of nodes Q such that (i) no action of l or of any node in Q can become forever enabled and never executed, (ii) every message sent between l and the nodes in Q is eventually delivered, and (iii) eventually, no node different from l tries to propose values. For comparison, [Lamport 2006] contains an informal proof of about a page of the same property under similar assumptions.

Practical systems use more complex protocols in the Paxos family such as Multi-Paxos [Lamport 2001], which allows a set of nodes to agree on a growing log (i.e. a sequence of values). Under the Paxos fairness assumptions, we prove that every position in the log is eventually agreed upon.

An important aspect of distributed systems is their dynamic nature: nodes crash and are replaced, and participants may arrive or leave the system dynamically. In such a dynamic environment, it is necessary to be able to reconfigure the set of participants of a protocol. Stoppable Paxos [Lamport et al. 2008, 2010] extends Multi-Paxos with the ability to stop all participating nodes, which all terminate with the same final log. The set of participants can then be reconfigured, after which log replication can resume using a new incarnation of Stoppable Paxos. Compared to alternative approaches to reconfiguration, Stoppable Paxos has both practical advantages (e.g. it does not limit parallelism in the absence of reconfiguration) and aesthetic advantages [Lamport et al. 2010]. However, it is one of the most intricate protocol in the Paxos family, exhibiting a dependency between ballots and instances not present in other Paxos variants. As admitted by [Lamport et al. 2008], "getting the details right was not easy". We model a single incarnation of Stoppable Paxos (thus the set of participants is fixed) and we prove that, under the Paxos fairness assumptions, for every position i in the log, eventually either a value is agreed upon or Stoppable Paxos stops with a final log of length strictly smaller than i. Our proof is the first mechanically-checked liveness proof of Stoppable Paxos. Under the same Paxos fairness assumptions, [Lamport et al. 2008] give an informal but detailed proof of the liveness property of Stoppable Paxos in about 3 pages of temporal-logic reasoning. Compared to this informal proof, proving liveness of Stoppable Paxos with our approach is more succinct and seems less tedious.

Our proofs are based on the IVy models of [Padon et al. 2017], with the addition of the temporal specification in FO-LTL and invariants for proving the safety of the system after the reduction. In all three cases, when applying the liveness-to-safety reduction, we exploit the fact that the set of nodes is finite (albeit unbounded) by initially adding all nodes to the relation d. We also use derived relations to increase the precision of the abstraction, e.g., by projecting away the value component of the relation modeling proposals in order to track whether some value has been proposed despite that value not being in the footprint at the freeze point. The proof requires only a single level.

7.2 Discussion of User Experience

Verifying the safety of the transition system resulting from the reduction. An important aspect of our approach is the effort required from the user to prove the safety of the reduced system. Our experience has been positive: in all of the examples we considered, we managed to find inductive invariants for the reduced systems with reasonable effort, comparable to the effort required for proving safety of the original system.

Recall that the vocabulary of the reduced system consists of the vocabulary of the original system and the vocabulary of the temporal property and the monitor. Existing invariants (from the safety proof) of the original system can be reused unchanged, and then additional invariants are needed to prove acyclicity. Thinking of these invariants is requires the user to understand the meaning of the new relations. This can be viewed as a different way of thinking about termination, and in our experience, when the user gets used to this way of thinking, finding invariants that prove acyclicity is comparable to finding usual safety invariants (which is indeed a creative task that requires a sophisticated user). During this process, IVy's feedback in the form of graphically displayed counterexamples to induction proved helpful, in the same way that it helps to find a usual safety invariant.

The invariant for the reduced system of the ticket protocol (Section 4.3) gives a good sense of the style (and complexity) of the invariants used to prove acyclicity. The form of this invariant is representative of the other examples we considered: the invariant is composed of parts that assert that a and d are large enough (the abstraction is precise enough), and parts that assert the existence of a difference between the current state and the saved state.

Fig. 10 lists for each example the number of conjectures needed to prove the safety of the original system ($\mathbf{C}_s$), the number of additional conjectures that only refer to the vocabulary of the original system ($\mathbf{C}_e$), and the number of conjectures that relate symbols of the original system to symbols of the monitor (($\mathbf{C}_m$). Each conjecture is a conjunct in the inductive invariant, and they are the "mental building blocks" of the invariant (most conjectures are expressed in one line). We thus use conjectures as a measure of the difficulty of finding inductive invariants. As the figure shows, the number of conjectures needed to prove liveness is $2 - 4$ times higher than the number required to prove safety. However, many of these conjectures are quite repetitive, and we found the effort required to prove liveness comparable to the effort required to prove safety.

Understanding that a nesting structure is needed. Another interesting point that requires sophistication from the user is the realization that a nesting structure is needed. This is somewhat analogous to the realization that a lexicographic ranking function is needed rather than a simple one in traditional termination proofs.

Our experience and intuition is that a nesting structure is needed when the transition system has an implicit "nested loop" structure. An indication of this is usually the fact that an important object is "missing" from the abstraction, as seen in a counterexample to induction. This intuition is demonstrated in Section 5.2 for the alternating bit protocol. While the alternating bit protocol does not syntactically contain a nested loop structure, there is an implicit nested loop present in the traces of the system: the inner loop transmits a single value (by retransmitting data and ack messages), and the outer loop iterates through the array of values. The verification of the TLB protocol contains more examples of the same flavor, where in the "outer loop" a thread is waiting for another thread to free a lock and in the "inner loop" the other thread checks that all other threads have set a flag.

We note that when attempting to prove such a system without a suitable nesting structure, the user may try to find an inductive invariant for the reduced system (which is actually not safe), and during this process the counterexamples to induction provided by IVy are helpful in assisting the user to realize that a nesting structure is needed (in a way which is similar to how counterexample to induction can help detect a bug).

8 RELATED WORK

There is an enormous body of literature on automatic or deduction-style temporal reasoning; see, e.g., [Dixon et al. 2008] or [Gotsman et al. 2009] and pointers therein. These works share common goals with our work (and some also first-order logic), but are mostly orthogonal to the topic of our work, which is the reduction of liveness to safety.

[Fang et al. 2006] prove liveness properties of parameterized systems by a liveness-to-safety reduction, and by applying the method of invisible invariants [Pnueli et al. 2001] to prove the resulting safety properties. They handle "bounded response" properties of the form $\Box (q \to \Diamond r)$, where a global bound exists on the number of rounds between q and r. In the perspective of response properties, our approach can handle cases in which there is no global bound on the number of rounds between q and r. Dynamic abstraction allows the bound to be dynamic, i.e., to be determined at the point where q happens, while a nesting structure can handle cases where no finite (dynamic) bound exists.

Liveness-to-safety reductions based on acyclicity in the style of [Biere et al. 2002] have been extended in [Schuppan and Biere 2006] to the settings of regular model checking, push-down systems, and timed automata. A contribution of our work is that we show how to extend acyclicity to the setting of first-order logic transition systems, which is beyond the above mentioned models. A key difference is that for regular model checking, push-down systems and timed automata, a dynamic abstraction (and the notion of a freeze point) is not needed. Instead, these restricted formalisms admit a fixed abstraction which makes cycle detection sound (e.g., since the set of states reachable from any initial state is finite).

In [Daniel et al. 2016], liveness of infinite-state systems is automatically proven by implicit predicate abstraction combined with well-founded relations. While their approach can automatically handle some infinite-state systems, it ultimately relies on a finite-state abstraction for the liveness-to-safety reduction, whereas our approach can handle systems in which no finite abstraction can be used to prove liveness.

Another interesting formalism for temporal verification of infinite-state systems is rewriting logic [Meseguer 1992, 2012], and the linear temporal logic of rewriting (LTLR) [Meseguer 2008]. This formalism, based on term rewriting, is extremely expressive, and can also express unbounded-parallelism. Indeed, our use of free variables in expressing infinitely many fairness constraints is analogous to rewriting logic's notion of localized and parameterized fairness constraints. Techniques for model checking temporal properties of rewrite systems have been developed in [Bae and Meseguer 2011, 2014, 2015]. Other than the differences in underlying formalism (term rewriting vs. first-order transition systems), these techniques differ from our work in that they use abstraction and simulation (via rewriting logic's narrowing) to obtain a symbolic state space that is finite (or with finitely many states reachable from any initial state). This is in contrast to our notion of dynamic abstraction and freezing. An interesting direction for future work could be to adapt our dynamic abstraction and acyclicity condition to the setting of rewriting logic.

We next discuss the reduction of liveness to safety for general programs which relies on the concept of *transition invariants* from [Podelski and Rybalchenko 2004b, 2011], a concept which originates from size-change analysis [Ben-Amram 2002; Lee et al. 2001]. Here, the safety property is the validity of a transition invariant. The burden of finding a ranking function for the input program is thus shifted to the burden of finding an inductive invariant for the input program. The shift of burden has proven useful in many approaches to proving program termination and liveness for many classes of programs; see, e.g., [Cook et al. 2007, 2006; Kroening et al. 2010; Kuwahara et al. 2014; Lee et al. 2012; Murase et al. 2016; Pnueli et al. 2005]. Since the transition invariant is translated from a set of ranking functions, one still has to find ranking functions. For arithmetic programs, this is in general considered a minor task, mostly because the ranking functions can be derived from automatic termination proofs for comparatively small programs of a specific form [Podelski and Rybalchenko 2004a]. However, these methods do not apply directly to systems like distributed protocols with states of a rich structure as considered by our work.

In the approach of *trace abstraction* for proving program termination and general LTL properties in [Dietsch et al. 2015; Heizmann et al. 2014], a set of ranking functions (which is constructed for a set of small programs of a specific form, as in the approach discussed above) is here translated to a set of finite-state Büchi automata (and not to a transition invariant). We then need to check the inclusion between Büchi automata (instead of checking the validity of a transition invariant). The inclusion check gets reduced to checking emptiness of a Büchi automaton, and thus to checking repeated reachability in a finite-state graph.

The advertised goal of the work in [Farzan et al. 2016] is a liveness-to-safety reduction for the verification of concurrent programs with an unbounded number of threads (the ticket protocol is given as a motivating example). Following the approach of *trace abstraction* as in [Dietsch et al. 2015; Heizmann et al. 2014], a set of ranking functions is now translated to a class of (infinite-state) Büchi automata (over an infinite alphabet) for which non-emptiness cannot be reduced to repeated reachability. Again, since the approach relies on ranking functions, it is not clear how the approach can be applied directly to systems like distributed protocols handled by our approach.

The works of [Abdulla et al. 2006; Cousot and Cousot 2012; Urban and Miné 2017] are related to our work in that it also avoids the constraint-based synthesis of ranking functions, replacing it essentially by new forms of widening in a CTL-style backwards fixpoint iteration. The approaches have not yet been extended to deal with general liveness and fairness and to deal with the data structures typically found in distributed protocols. In the termination analysis for heap-manipulating programs presented in [Manevich et al. 2016], the use of ranking functions is avoided by a novel form of reasoning over sets of nodes, sets which are *a priori* known to be finite. However, as we have seen, for liveness of distributed systems the key is to identify when to fix a finite set, a problem which does not exists when considering heap-manipulating programs

The work in [Bloem et al. 2015; Konnov et al. 2017, 2015a,b] addresses the problem of verifying liveness by using a restricted language, the formalism of Threshold Automata, for specifying distributed algorithms operating in a partially synchronous communication mode. This restriction allows them to derive decision procedures based on cutoff theorems, both for safety and liveness. The derivation of the bounds on the lengths of counterexamples seems specific to the formalism, and not related to the implicit bounds in our setting. It is not clear whether the formalism can capture the complexity of the distributed protocols considered in our work.

To summarize, none of the existing approaches proposes a liveness-to-safety reduction that addresses the problem of verifying liveness for distributed protocols and other systems modeled with first-order logic. In particular, our notion of dynamic finite abstraction and our acyclicity condition differ from existing works that either use ranking functions, or a fixed abstraction (with finitely many states reachable from any initial state). Furthermore, none of the existing approaches has demonstrated its practical potential on verification problems at the scale of the examples in our practical evaluation.

9 CONCLUSION

This work presents a novel way to automatically reduce temporal verification of infinite-state systems, expressed in first-order logic, to safety verification. Our approach exploits the first-order formalism in two ways: (i) to define a uniform notion of an *abstract fair cycle* which is the key idea of the reduction, and (ii) to verify the safety property obtained by the reduction using first-order verification conditions. We used our approach to verify temporal properties of several challenging distributed protocols. In these examples, after applying the reduction, we interactively found inductive invariants for proving safety. In most cases we obtained verification conditions in EPR.

In this paper we used uninterpreted first-order logic. However, the liveness-to-safety reduction based on the acyclicity condition, as well as its realization in first-order logic, do not depend on the fact that all relations are uninterpreted. In principle, the same reduction can be used in the presence of interpreted relations and background theories. In this case, it may be beneficial to adjust the definition of the footprint in a sound way that will improve the proof power of the reduction (e.g., in arithmetic of natural numbers, we may add all elements smaller than any constant, and not just the constant itself). In such a scenario, the resulting system would also contain interpreted relations, and its safety can be verified using an inductive invariant, checked by an SMT solver for the theories involved.

Our approach also opens the path to using techniques for automatic invariant inference in first-order logic (e.g., [Karbyshev et al. 2015; Sagiv et al. 2002]) for temporal verification. While automatically inferring the inductive invariants for the examples considered here is beyond the scope of existing techniques, future techniques may allow fully automated temporal verification.

ACKNOWLEDGMENTS

We thank Amir Ben-Amram, Neil Immerman, Ken McMillan, Alexander Nutz, Alexander Rabinovich, our shepherd Grigore Rosu, the anonymous referees and the anonymous artifact evaluation referees for insightful comments which improved this paper. Padon was supported by Google under a PhD fellowship. Padon and Sagiv were supported by the European Research Council under the European Union's Seventh Framework Program (FP7/2007–2013) / ERC grant agreement no. [321174-VSSC]. This research was partially supported by Len Blavatnik and the Blavatnik Family foundation, and by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University. This material is based upon work supported by the National Science Foundation under Grant No. 1655166 and by the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259.

REFERENCES

Martín Abadi. 1989. The Power of Temporal Proofs. *Theor. Comput. Sci.* 65, 1 (1989), 35–83. https://doi.org/10.1016/0304-3975(89)90138-2

Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. 2006. Proving Liveness by Backwards Reachability. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 4137. Springer, 95–109.

Kyungmin Bae and José Meseguer. 2011. State/Event-Based LTL Model Checking under Parametric Generalized Fairness. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–148. https://doi.org/10.1007/978-3-642-22110-1_11

Kyungmin Bae and José Meseguer. 2014. Infinite-State Model Checking of LTLR Formulas Using Narrowing. In *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*. 113–129. https://doi.org/10.1007/978-3-319-12904-4_6

Kyungmin Bae and José Meseguer. 2015. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming* 99, Supplement C (2015), 193 – 234. https://doi.org/10.1016/j.scico.2014.02.006 Selected

Papers from the Ninth International Workshop on Rewriting Logic and its Applications (WRLA 2012).

Amir M. Ben-Amram. 2002. General Size-Change Termination and Lexicographic Descent. In *The Essence of Computation (Lecture Notes in Computer Science)*, Vol. 2566. Springer, 3–17.

Armin Biere, Cyrille Artho, and Viktor Schuppan. 2002. Liveness Checking as Safety Checking. *Electr. Notes Theor. Comput. Sci.* 66, 2 (2002), 160–177.

D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. 1989. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*. ACM, New York, NY, USA, 113–122. https://doi.org/10.1145/70082.68193

Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00658ED1V01Y201508DCT013

B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. 2007. Proving that programs eventually do something good. In *POPL*, Martin Hofmann and Matthias Felleisen (Eds.). 265–276.

B. Cook, A. Podelski, and A. Rybalchenko. 2006. Termination proofs for systems code. In *PLDI*. 415–426.

Jonathan Corbet. 2008. Ticket spinlocks. https://lwn.net/Articles/267968/. (2008).

P. Cousot and R. Cousot. 2012. An abstract interpretation framework for termination. In *POPL*. 245–258.

Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. 2016. Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 271–291. https://doi.org/10.1007/978-3-319-41528-4_15

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.

Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. 2015. Fairness Modulo Theory: A New Approach to LTL Software Model Checking. In *CAV (Lecture Notes in Computer Science)*, Vol. 9206. Springer, 49–66.

Clare Dixon, Michael Fisher, Boris Konev, and Alexei Lisitsa. 2008. Practical First-Order Temporal Reasoning. In *TIME*. IEEE Computer Society, 156–163.

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. *ACM SIGPLAN Notices* 51, 1 (2016), 400–415.

Yi Fang, Kenneth L. McMillan, Amir Pnueli, and Lenore D. Zuck. 2006. Liveness by Invisible Invariants. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006. (Lecture Notes in Computer Science)*, Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge (Eds.), Vol. 4229. Springer, 356–371. https://doi.org/10.1007/11888116_26

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving Liveness of Parameterized Programs. In *LICS*. ACM, 185–196.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Proving that non-blocking algorithms don't block. In *POPL*. 16–28. https://doi.org/10.1145/1480881.1480886

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*. 1–17.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. *CoRR* abs/1405.4189 (2014).

Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. 2017. Thread modularity at many levels: a pearl in compositional verification. In *POPL*. ACM, 473–485.

Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2015. Property-Directed Inference of Universal Invariants or Proving Their Absence. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 583–602. https://doi.org/10.1007/978-3-319-21690-4_40

Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017. A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant Distributed Algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 719–734.

Igor Konnov, Helmut Veith, and Josef Widder. 2015a. SMT and POR Beat Counter Abstraction: Parameterized Model Checking of Threshold-Based Distributed Algorithms. In *Computer Aided Verification*. Springer, Cham, 85–102.

Igor V. Konnov, Helmut Veith, and Josef Widder. 2015b. What You Always Wanted to Know About Model Checking of Fault-Tolerant Distributed Algorithms. In *Perspectives of System Informatics - 10th International Andrei Ershov Informatics*

Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers (Lecture Notes in Computer Science), Manuel Mazzara and Andrei Voronkov (Eds.), Vol. 9609. Springer, 6–21. https://doi.org/10.1007/978-3-319-41579-6_2

Konstantin Korovin. 2008. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. 292–298.

Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *CAV (Lecture Notes in Computer Science)*, Vol. 6174. Springer, 89–103.

Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *ESOP (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 392–411.

Leslie Lamport. 1974. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM* 17, 8 (Aug. 1974), 453–455. https://doi.org/10.1145/361082.361093

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport. 2001. Paxos Made Simple. (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2008. *Stoppable Paxos*. Technical Report. TechReport, Microsoft Research. https://www.microsoft.com/en-us/research/publication/stoppable-paxos/

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a State Machine. *SIGACT News* 41, 1 (03 2010), 63–73.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *POPL*. ACM, 81–92.

Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. 2012. Termination Analysis with Algorithmic Learning. In *CAV (Lecture Notes in Computer Science)*, Vol. 7358. Springer, 88–104.

Roman Manevich, Boris Dogadov, and Noam Rinetzky. 2016. From Shape Analysis to Termination Analysis in Linear Time. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 426–446.

Zohar Manna and Amir Pnueli. 1983. Verification of Concurrent Programs: A Temporal Proof System. In *Foundations of Computer Science: Distributed Systems*, J. W. de Bakker and J. van Leeuwen (Eds.). Mathematisch Centrum, Amsterdam, 163–255.

Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems - safety*. Springer.

Kenneth L. McMillan. 2016. Modular specification and verification of a cache-coherent interface. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 109–116. https://doi.org/10.1109/FMCAD.2016.7886668

José Meseguer. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96, 1 (1992), 73–155.

José Meseguer. 2008. The Temporal Logic of Rewriting: A Gentle Introduction. In *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday (Lecture Notes in Computer Science)*, Pierpaolo Degano, Rocco De Nicola, and José Meseguer (Eds.), Vol. 5065. Springer, 354–382. https://doi.org/10.1007/978-3-540-68679-8_22

José Meseguer. 2012. Twenty years of rewriting logic. *J. Log. Algebr. Program.* 81, 7-8 (2012), 721–781. https://doi.org/10.1016/j.jlap.2012.06.003

Akihiro Murase, Tachio Terauchi, Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2016. Temporal verification of higher-order functional programs. In *POPL*. ACM, 57–68.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos Made EPR: Decidable Reasoning About Distributed Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (Oct. 2017), 31 pages. https://doi.org/10.1145/3140568

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 614–630.

Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. 2010. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *J. Autom. Reasoning* 44, 4 (2010), 401–424.

Amir Pnueli, Andreas Podelski, and Andrey Rybalchenko. 2005. Separating Fairness and Well-Foundedness for the Analysis of Fair Discrete Systems. In *TACAS (Lecture Notes in Computer Science)*, Vol. 3440. Springer, 124–139.

Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Tiziana Margaria and Wang Yi (Eds.), Vol. 2031. Springer, 82–97. https://doi.org/10.1007/3-540-45319-9_7

Amir Pnueli and Elad Shahar. 2000. Liveness and Acceleration in Parameterized Verification. In *CAV (Lecture Notes in Computer Science)*, Vol. 1855. Springer, 328–343.

Andreas Podelski and Andrey Rybalchenko. 2004a. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI (Lecture Notes in Computer Science)*, Vol. 2937. Springer, 239–251.

Andreas Podelski and Andrey Rybalchenko. 2004b. Transition Invariants. In *LICS*. IEEE Computer Society, 32–41.

Andreas Podelski and Andrey Rybalchenko. 2011. Transition Invariants and Transition Predicate Abstraction for Program Termination. In *TACAS (Lecture Notes in Computer Science)*, Vol. 6605. Springer, 3–10.

F. Ramsey. 1930. On a problem in formal logic. In *Proc. London Math. Soc.*

Alexandre Riazanov and Andrei Voronkov. 2002. The Design and Implementation of VAMPIRE. *AI Commun.* 15, 2,3 (Aug. 2002), 91–110. http://dl.acm.org/citation.cfm?id=1218615.1218620

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. https://doi.org/10.1145/514188.514190

Viktor Schuppan and Armin Biere. 2006. Liveness Checking as Safety Checking for Infinite State Spaces. *Electr. Notes Theor. Comput. Sci.* 149, 1 (2006), 79–96.

Caterina Urban and Antoine Miné. 2017. Inference of ranking functions for proving temporal properties by abstract interpretation. *Computer Languages, Systems & Structures* 47 (2017), 77–103.

M.Y. Vardi and P. Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. 1st Symp. on Logic in Computer Science*. Cambridge, 332–344. http://www.cs.rice.edu/~vardi/papers/lics86.pdf.gz

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS Version 3.5. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*. 140–145.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368.

Pierre Wolper. 2000. Constructing Automata from Temporal Logic Formulas: A Tutorial. In *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures (Lecture Notes in Computer Science)*, Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen (Eds.), Vol. 2090. Springer, 261–277. https://doi.org/10.1007/3-540-44667-2_7