Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions

JIANAN YAO, Columbia University, USA
RUNZHOU TAO, Columbia University, USA
RONGHUI GU, Columbia University, USA
JASON NIEH, Columbia University, USA

Distributed protocols have long been formulated in terms of their safety and liveness properties. Much recent work has focused on automatically verifying the safety properties of distributed protocols, but doing so for liveness properties has remained a challenging, unsolved problem. We present LVR, the first framework that can mostly automatically verify liveness properties for distributed protocols. Our key insight is that most liveness properties for distributed protocols can be reduced to a set of safety properties with the help of ranking functions. Such ranking functions for practical distributed protocols have certain properties that make them straightforward to synthesize, contrary to conventional wisdom. We prove that verifying a liveness property can then be reduced to a simpler problem of verifying a set of safety properties, namely that the ranking function is strictly decreasing and nonnegative for any protocol state transition, and there is no deadlock. LVR automatically synthesizes ranking functions by formulating a parameterized function of integer protocol variables, statically analyzing the lower and upper bounds of the variables as well as how much they can change on each state transition, then feeding the constraints to an SMT solver to determine the coefficients of the ranking function. It then uses an off-the-shelf verification tool to find inductive invariants to verify safety properties for both ranking functions and deadlock freedom. We show that LVR can mostly automatically verify the liveness properties of several distributed protocols, including various versions of Paxos, with limited user guidance.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Automated reasoning**; • **Networks** → **Protocol testing and verification**; • **Software and its engineering** → *Formal software verification*; • **Security and privacy** → Distributed systems security.

Additional Key Words and Phrases: distributed protocols, liveness reasoning, ranking function synthesis

ACM Reference Format:
Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2024. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 35 (January 2024), 32 pages. https://doi.org/10.1145/3632877

1 INTRODUCTION

Distributed systems are of critical importance in computing infrastructure for serving billions of people, but they remain complex and difficult to build and maintain correctly. Formal verification has gradually become more useful to provably rule out bugs in such systems. A key aspect of such a correctness proof is formally verifying that desired safety and liveness properties hold for the distributed protocol. A safety property specifies what a protocol must not do, while a

Authors' addresses: Jianan Yao, Columbia University, New York, NY, USA, jianan@cs.columbia.edu; Runzhou Tao, Columbia University, New York, NY, USA, runzhou.tao@columbia.edu; Ronghui Gu, Columbia University, New York, NY, USA, ronghui.gu@columbia.edu; Jason Nieh, Columbia University, New York, NY, USA, nieh@cs.columbia.edu.

ACM 2475-1421/2024/1-ART35
https://doi.org/10.1145/3632877

liveness property specifies what a protocol must eventually do. For example, for a distributed lock protocol [Raymond 1989], its safety property is that no two clients in the system hold a lock simultaneously, while its liveness property is that a client that is waiting to acquire a lock will eventually acquire the lock.

Because proof effort has been a significant barrier to further adoption of formal verification, much recent work has focused on tooling to automate various aspects of verifying the correctness of distributed protocols [Hance et al. 2021; Koenig et al. 2022; Ma et al. 2022, 2019; Padon et al. 2016; Wilcox et al. 2018; Yao et al. 2022]. For example, tools have been developed based on SMT (satisfiability modulo theories) solvers to automatically find inductive invariants that imply and verify safety properties for distributed protocols [Hance et al. 2021; Koenig et al. 2022; Ma et al. 2019; Yao et al. 2022]. While much progress has been made in automatically verifying safety properties, doing so for liveness properties has remained an open problem.

Liveness properties are much harder to verify than safety properties; in fact, the problem is undecidable in the general case. Consider a distributed protocol as a specification of the behavior of a state machine. Proving safety can be done by only reasoning about two system states at a time—if any single transition from one state to another preserves the safety property, then by induction, the safety property holds for all state transitions. However, proving liveness, the notion that the system must eventually do something, potentially requires reasoning about an infinite series of system states. This kind of reasoning involving the additional dimension of time is problematic to use with SMT solvers, which will frequently time out, leaving developers guessing as to whether the desired liveness property holds or not.

Previous approaches to verifying liveness properties for distributed protocols require extensive manual effort. IronFleet [Hawblitzel et al. 2015a] was the first to verify the liveness properties of distributed protocols with SMT solvers, but required thousands of lines of user inputs in Dafny for not just the protocol specification but its implementation as well to complete the proof. Liveness properties were expressed in temporal logic, which was problematic to use with SMT solvers without causing them to time out. To avoid timeouts, IronFleet employed various workarounds that required extensive human effort to tune triggers to tell the underlying SMT solver what to do and what not to do. Dynamic abstraction [Padon et al. 2017, 2021] reduces a liveness property of a distributed protocol to a safety property of another protocol that extends the original one, but the extended protocol is more complex and involves a much larger number of states. Unfortunately, proving the safety property of the extended protocol can require finding dozens of additional invariants, which must be done manually since it is well beyond the scope of existing automated techniques. The resulting proof structure with many invariants unrelated to the original protocol is also not human-readable.

We present LVR (Liveness Verification with Ranking functions), the first framework that can mostly automate the verification of liveness properties for distributed protocols. Our key insight is that most liveness properties for distributed protocols can be reduced to a set of safety properties with the help of ranking functions, which can be automatically synthesized using an SMT solver with first-order logic and nonlinear integer arithmetic. While conventional wisdom claims that synthesizing ranking functions is in general hard for transition systems [Padon et al. 2017], we observe that in the context of distributed protocols, ranking functions have certain properties and formulations that make them much easier to synthesize and analyze than previously believed. Ranking functions of distributed protocols can often be synthesized as polynomial functions of integer variables from the protocol specification. We prove that verifying a liveness property can then be reduced to a simpler problem of verifying a set of safety properties, namely that the ranking function is strictly decreasing and nonnegative between any two protocol states for any state transition, the protocol never deadlocks, and the protocol can only end in a "good" state.

LVR automatically synthesizes a ranking function by combining non-constant integer variables from the protocol specification, with each variable assigned a weight. LVR first analyzes the specification to determine lower and upper bounds for each variable, as well as *deltas* that indicate how each variable can change as a result of any protocol state transition. This results in a set of constraints on the weights, where each weight is expressed as a function of constant integer variables from the protocol specification. These weight constraints result in a set of linear inequalities constraining the coefficients of constant integer variables, which are then fed to an SMT solver to solve for the coefficients to synthesize the ranking function. A key factor is that the shape of the ranking function synthesized by LVR allows the constraints to be effectively solved by the SMT solver. We also introduce a provably sound tiering mechanism to decompose the synthesis and validation of high-order ranking functions, enabling LVR to scale to complex protocols.

LVR ensures that each step of the synthesis process is guarded by the SMT solver. For example, the bounds and deltas are treated as safety properties, which are verified by finding inductive invariants using an off-the-shelf invariant inference tool based on an SMT solver. Similarly, once the ranking function has been synthesized, we also use an SMT solver to verify that the function is strictly decreasing and nonnegative. The absence of deadlock is also a safety property, which is verified using an invariant inference tool based on an SMT solver. LVR leverages mypyvy [Wilcox et al. 2018] and its SMT solvers, as well as P-FOL-IC3 [Koenig et al. 2022] for finding inductive invariants to verify safety properties.

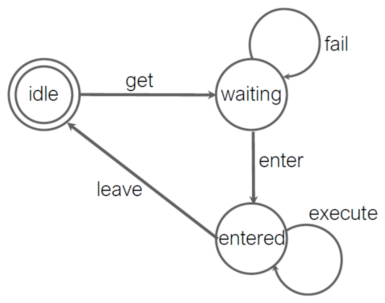
While LVR is mostly automated, some user guidance may be needed as part of the verification process. This can include specifying additional variables not in the protocol specification, tightening automatically inferred bounds and deltas, and writing invariants if the off-the-shelf invariant inference tool fails. Most user guidance only requires human intuition based on an understanding of the protocol being verified and is straightforward to provide.

We use LVR to automatically verify several distributed protocols, including several versions of Paxos, with only limited user guidance. These protocols previously could only be verified with significant manual effort [Hawblitzel et al. 2015a; Padon et al. 2017; Wilcox et al. 2015]. Furthermore, we show that the resulting proof using LVR is easier to read and understand, and requires much fewer invariants to consider compared to previous approaches. Although LVR is not guaranteed to complete liveness verification since the problem in general is undecidable, these results show that LVR works well for practical distributed protocols. These results are the first to demonstrate that the verification of liveness properties for distributed protocols can be mostly automated without significant manual effort.

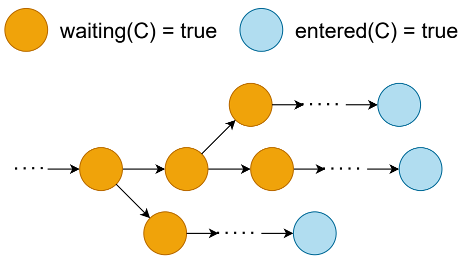
2 LVR OVERVIEW

We use the ticket lock protocol, taken from the Ivy repository [Padon 2021], as a running example to show how LVR works. The ticket lock protocol uses tickets to control access to a shared resource on a server, where each client takes a ticket with a number, and only the client holding the current number being served can enter the critical section. To acquire the lock, a client requests a new ticket and waits until the ticket being served reaches its own, at which point it can enter the critical section. The ticket being served increments when a client releases the lock.

Figure 1a visualizes the ticket lock protocol as a state transition system specified by a state space and a set of transitions. Executing the protocol involves applying a series of transitions on an initial state to update the system state. The state of each client is either `idle` (not having a ticket), `waiting` (having a ticket and waiting for the lock), or `entered` (in the critical section). The protocol defines five transitions (`get`, `enter`, `execute`, `leave`, and `fail`) to update a client's state. The protocol also maintains two global variables `now` (current ticket being served) and `next` (next ticket to allocate)



(a)



(b)

Fig. 1. (a) State transition diagram for a single client in ticket lock protocol; (b) Liveness property of ticket lock. A circle represents a protocol state and an arrow represents a transition. For any client C, given a state where `waiting(C)` $=$ *true*, no matter how the system nondeterministically evolves, there will be a future state where `entered(C)` $=$ *true*.

shared among clients, and a local variable `myt` (a client's own ticket) for each client. For simplicity, we assume `now` and `next` can be accessed atomically by each client.

Liveness property. The liveness property for the ticket lock protocol ensures that all clients waiting for the lock will eventually enter the critical section, which can be expressed in first-order linear temporal logic as follows

$$\forall \texttt{C: client.} \square(\texttt{waiting(C)} \longrightarrow \lozenge\texttt{entered(C)}), \tag{1}$$

where $\square$ denotes *globally* and $\lozenge$ denotes *finally*. Figure 1b gives an illustration of this property through the lens of execution traces.

Fairness assumptions. The liveness property does not hold unconditionally; it requires *fairness assumptions*. For the ticket lock protocol, if the server never schedules some client, or the client in the critical section does not exit, the entire system will be starved. To prove liveness, the protocol makes two fairness assumptions:

Every client is scheduled infinitely often. (2)

Any client in the critical section can only execute a finite number of steps before leaving. (3)

LVR workflow. Figure 2 shows the workflow of using LVR to verify the liveness property of the ticket lock protocol. Verifying the liveness of the ticket lock protocol means proving the liveness property Eq. (1) using the fairness assumptions Eq. (2) and (3). The second fairness assumption Eq. (3) is straightforward and typically embedded in the protocol specification [Padon 2021] by counting and limiting execution steps. We focus on the first fairness assumption Eq. (2), which is usually formalized as a linear temporal logic formula:

$$\forall \texttt{C: client.} \square(\lozenge\texttt{scheduled(C)}). \tag{4}$$

SMT solvers do not directly support linear temporal logic notation, but Eq. (4) can be translated into an equivalent first-order logic formula usable with SMT solvers [Hawblitzel et al. 2015a]:

$$\forall \texttt{C: client}, \forall \texttt{T}_1\texttt{: time}, \exists \texttt{T}_2\texttt{: time. T}_1 \le \texttt{T}_2 \land \texttt{scheduled(C)}|_{\texttt{T}_2}, \tag{5}$$

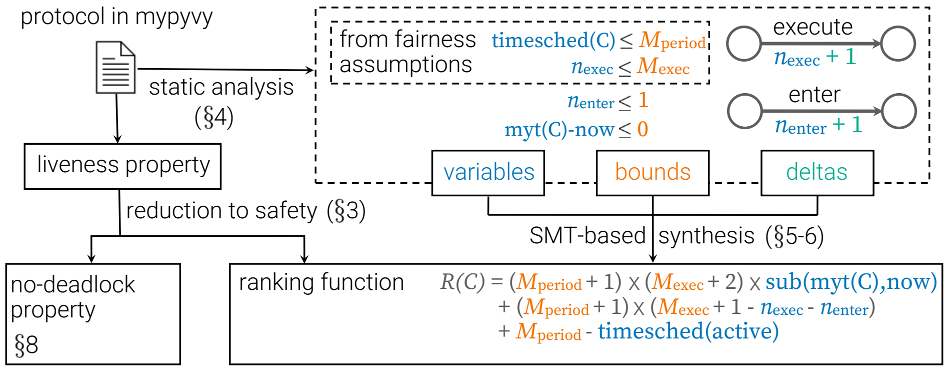

protocol in mypyvy
static analysis (§4)
liveness property
reduction to safety (§3)
no-deadlock property §8
from fairness assumptions
$\mathrm{timesched(C)} \leq M_{\mathrm{period}}$
$n_{\mathrm{exec}} \leq M_{\mathrm{exec}}$
$n_{\mathrm{enter}} \leq 1$
$\mathrm{myt(C)\text{-}now} \leq 0$
execute
$n_{\mathrm{exec}} + 1$
enter
$n_{\mathrm{enter}} + 1$
variables
bounds
deltas
SMT-based synthesis (§5-6)
ranking function
$R(C) = (M_{\mathrm{period}} + 1) \times (M_{\mathrm{exec}} + 2) \times \mathrm{sub(myt(C),now)} + (M_{\mathrm{period}} + 1) \times (M_{\mathrm{exec}} + 1 - n_{\mathrm{exec}} - n_{\mathrm{enter}}) + M_{\mathrm{period}} - \mathrm{timesched(active)}$

Fig. 2. LVR workflow to verify the liveness property for ticket lock protocol.

where `time` is logical time counting steps rather than wall-clock time. Eq. (5) states that for any client, at any time $\mathtt{T}_1$, there exists a future time $\mathtt{T}_2$ at which the client is scheduled. Although Eq. (5) is not hard for humans to understand, such a formula poses two key challenges for automating liveness proofs. First, the formula has quantifier alternation, meaning that ∀ and ∃ quantifiers are mixed together, which is known to frequently cause SMT solvers to timeout [Yao et al. 2022]. Second, although the formula implies a bound on the time, it is not explicitly stated in the formula. However, synthesizing a ranking function to prove liveness properties can be complex and difficult with only implicit bounds in fairness assumptions, as then the bounds cannot be used explicitly in the ranking function.

Encoding fairness. We make two key insights to enable LVR to use SMT solvers and ranking functions to prove liveness properties. First, we can eliminate quantifier alternation from fairness assumptions if the time implied by such assumptions can be captured in the protocol state itself. In this way, fairness assumptions no longer need to be formulated as whether or not some time exists because it necessarily exists as part of the protocol state. Second, since fairness assumptions necessarily imply some bound on time, we can make explicit its existence by introducing an abstract bound in the formula. The abstract bound is not fixed to any specific value, but simply provides a concrete declaration that a bound exists. Based on these insights, we can encode Eq. (5) by introducing and tracking a time `timesched(C)`, the time since client `C` was last scheduled, and an abstract bound M_{period}. `timesched(C)` is reset to 0 when `C` is scheduled. Then the fairness assumption is rewritten as:

$$\forall \mathtt{C:\ client},\ \forall \mathtt{T:\ time}.\ \mathtt{timesched(C)}|_{\mathtt{T}} \leq M_{\mathrm{period}}. \tag{6}$$

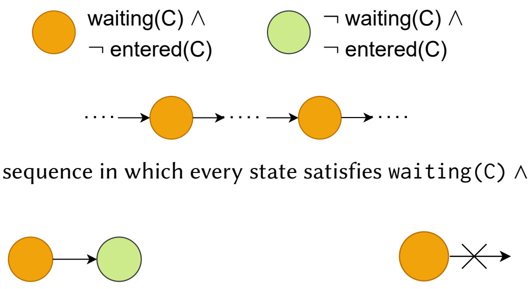
Eq. (6) is essentially equivalent to and expresses the same fairness assumption as Eq. (5). The only minor difference is the abstract bound is global instead of for each scheduling decision, but otherwise the intuition and expression of the fairness assumption remain the same. Section 7 provides further details. LVR can synthesize ranking functions based on Eq. (6).

Figure 3 shows the ticket lock protocol specification, using Eq. (6) at Line 51. Lines 2-7 specify the variables representing the protocol state. Lines 20-49 specify the five transitions the protocol can take. `get(c)` requires that client `c` is idle. `fail(c)` happens when client `c` does not hold the ticket being served `now` so that the client fails to enter the critical section. `enter(c)` requires client `c` is waiting and holds the ticket being served so the client can enter the critical section. `exec(c)` happens when client `c` executes an instruction in the critical section. `leave(c)` happens when client `c` leaves the critical section, incrementing the ticket number being served. Each transition executes

```
1  sort client, ticket
2  mutable relation idle(client), waiting(client), entered(client)
3  mutable constant now: ticket, next: ticket
4  mutable function myt(client): ticket
5  mutable constant n_exec: int
6  mutable function timesched(client): int
7  immutable constant M_exec: int, M_period: int
8  axiom M_exec > 0 & M_period > 0
9
10 init forall C. idle(C) & !waiting(C) & !entered(C)
11 init now = 0 & next = 0
12 init forall C. myt(C) = 0
13 init n_exec = 0
14 init forall C. timesched(C) = 0
15
16 procedure set_timesched(c:client)
17   timesched(c) = 0;
18   forall C. C != c -> timesched(c) = timesched(c) + 1;
19
20 transition get(c:client)
21   precondition idle(c);
22   idle(c) = false;
23   waiting(c) = true;
24   myt(c) = next;
25   next = next + 1;
26   set_timesched(c);
27
28 transition fail(c:client)
29   precondition waiting(c) & myt(c) != now;
30   set_timesched(c);
31
32 transition enter(c:client)
33   precondition waiting(c) & myt(c) = now;
34   waiting(c) = false;
35   entered(c) = true;
36   set_timesched(c);
37
38 transition execute(c:client)
39   precondition entered(c);
40   n_exec = n_exec + 1;
41   set_timesched(c);
42
43 transition leave(c:client)
44   precondition entered(c);
45   entered(c) = false;
46   idle(c) = true;
47   now = now + 1;
48   n_exec = 0;
49   set_timesched(c);
50
51 trusted invariant forall C. timesched(C) <= M_period
52 trusted invariant n_exec <= M_exec
```

Fig. 3. Ticket lock protocol specification. In mypyvy, "constant" represents a nullary relation or function, instead of describing immutability, and "&" stands for logical AND. For illustrative purposes, we show transitions in an imperative style, use an arithmetic "+" instead of a successor relation at Lines 25 and 47, and omit an ordering relation for tickets for brevity.

atomically and can only happen if its preconditions are satisfied. Transitions from different clients can interleave arbitrarily.



(a) An infinite sequence in which every state satisfies `waiting(C)` ∧ ¬`entered(C)`.

(b) A transition out of `waiting(C)` without `entered(C)`.

(c) A deadlock state where no transition can be taken.

Fig. 4. The three cases that the liveness proof needs to rule out.

Encoding liveness. Having reduced the fairness assumptions out of temporal logic, we consider how we can do the same with the liveness property, even though it may seem more entrenched in temporal logic. Instead of directly proving Eq. (1), it suffices to prove three simpler properties. If we call `waiting(C)` ∧ ¬`entered(C)` a "bad" state where a symbolic client `C` is waiting for the lock and has not entered the critical section, the three properties are: 1) the "bad" state will terminate after a finite number of transitions; 2) the system will not encounter deadlock and can always make transitions in the "bad" state; and 3) when the "bad" state ends, it must be in a "good" state where `C` enters the critical section, rather than ceasing to wait. Figure 4 gives an illustration of what executions are ruled out by each of the three properties.

The latter two are safety properties that do not involve infinite sequences of states and can be solved using automated invariant inference tools. The first property still involves infinite sequences of states but becomes a termination problem, which can be proved if we successfully find a *ranking function* to limit the number of steps before the termination, analogous to proving termination of loops and recursive functions [Ben-Amram and Genaim 2017; Cook et al. 2007a; Heizmann and Leike 2015; Neumann et al. 2020]. For a distributed protocol, a ranking function decreases after each protocol transition while remaining nonnegative. Both monotonically decreasing and nonnegative properties involve only two consecutive states and can be proved as safety properties. Section 3 formalizes this liveness-to-safety reduction.

Synthesizing ranking functions. Although finding a valid ranking function is generally hard, we observe that ranking functions for distributed protocols are usually not that complex. They can be expressed in polynomial form with integer-valued variables, making it possible to infer and verify them automatically. Based on this observation, LVR introduces an approach to synthesizing ranking functions by first enumerating what variables may appear in the ranking function, then solving for the coefficients for each variable.

It is natural to include variables in the ranking function from the original protocol specification, such as `myt(C)`, and the ones in the encoded fairness assumptions, such as n_{exec} and `timesched(C)`, as listed at Lines 4-6 in Figure 3 for the ticket lock protocol. Beyond these, LVR also introduces other counting variables related to the protocol's configuration, such as n_{enter} for the number of clients that are in the entered state for the ticket lock protocol.

If LVR can infer each variable's bounds and deltas, the task can be reduced to solving for coefficients to make the formula monotonically decreasing and nonnegative with the constraints of bounds and deltas, which can be efficiently solved by SMT solvers. The bounds can be constant integers or some constants that depend on the protocol's configuration. For example, the ticket lock protocol has $n_{\text{enter}} \leq 1$, which means that only at most one client can be in the critical section.

As shown in Figure 2, LVR infers bounds based on fairness assumptions, such as $n_{\text{exec}} \leq M_{\text{exec}}$ at Line 52 in Figure 3 for the ticket lock protocol, and reasoning regarding how protocol transitions affect variables, such as `timesched(C)` ≥ 0 for the ticket lock protocol. After finding bounds, LVR infers deltas of each integer variable by statically analyzing each transition. For example, LVR can infer that the delta of n_{enter} is 1 after each `enter` transition. Section 4 details the static analysis LVR employs to infer integer variables, bounds, and deltas. LVR will automatically verify the inferred bounds and deltas as safety properties.

Some limited user guidance may be needed, in terms of including additional variables or tightening the inferred bounds or deltas. For the ticket lock protocol, anyone who understands the protocol will recognize that there is an intuitive yet implicit notion of an `active` client that holds the ticket currently being served. This notion is needed for liveness reasoning, but is not explicitly declared as a variable in the protocol specification. As a result, user guidance is needed to provide an explicit variable denoting the `active` client. Section 9.2 quantifies the user guidance required to verify eight distributed protocols.

LVR then synthesizes a ranking function over integer variables that monotonically decreases on each transition and remains nonnegative. Nonnegativity defines constraints on variables' bounds, and monotonicity defines constraints on variables' deltas. These constraints are translated to inequalities over coefficients in the ranking function, and solved by an SMT solver. For ticket lock protocol, the ranking function inferred by LVR is:

$$\begin{aligned} R(\mathtt{C}) =& (M_{\text{period}} + 1) * (M_{\text{exec}} + 2) * sub(\mathtt{myt(C)}, \mathtt{now}) + \\ & (M_{\text{period}} + 1) * (M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}}) + \\ & M_{\text{period}} - \mathtt{timesched(active)}. \end{aligned} \tag{7}$$

After finding the ranking function, LVR uses the bounds and deltas to prove that $R(\mathtt{C})$ is nonnegative and monotonically decreasing after each transition as safety properties. Section 5 shows how LVR utilizes an SMT solver to synthesize ranking functions. Section 6 shows how LVR can scale ranking functions by tiering to handle complex protocols.

Finally, as shown in Figure 2, LVR proves the absence of deadlock in the protocol. Section 8 shows how LVR rules out deadlocks in the protocol using an automated invariant inference tool.

3 GENERAL REDUCTION OF LIVENESS PROPERTIES

We prove that our approach can be generalized to any protocol's liveness property, which states that something good will eventually happen after certain conditions are triggered. LVR considers any liveness property in the following form:

> For any value `V` in the given set of variables, if at any time the `trigger(V)` condition is satisfied, the `good(V)` condition will eventually be satisfied.

This can be expressed formally in first-order linear temporal logic as:

$$\forall \mathtt{V:\ var.} \square(\mathtt{trigger(V)} \longrightarrow \lozenge \mathtt{good(V)}). \tag{8}$$

Compared with the liveness property for ticket lock in Section 2, we can see that the `trigger` condition corresponds to `waiting(C)`, while the `good` condition corresponds to `entered(C)`.

Liveness properties of distributed protocols can naturally be expressed using Eq. (8). A network transmission protocol ensures that once a message is sent (the `trigger` condition), the message will eventually be received (the `good` condition). A consensus protocol ensures that a common value is eventually reached (the `good` condition), where the `trigger` condition can be a degenerate `True` in this case. We prove that verifying any liveness property in the form of Eq. (8) can be reduced to verifying four safety properties, as stated in the following theorem:

Theorem 1. *Let* $\textbf{old}(e)$ *denote the expression e's value in the previous state. If the following four safety properties hold for some ranking function* rank*,*

$$\forall \mathtt{V},\ \textbf{old}(\mathtt{trigger(V)} \land \neg \mathtt{good(V)}) \longrightarrow \mathtt{rank(V)} < \textbf{old}(\mathtt{rank(V)}) \tag{9}$$

$$\forall \mathtt{V},\ (\mathtt{trigger(V)} \land \neg \mathtt{good(V)}) \longrightarrow \mathtt{rank(V)} \geq 0 \tag{10}$$

$$\forall \mathtt{V},\ (\mathtt{trigger(V)} \land \neg \mathtt{good(V)}) \longrightarrow \exists \mathtt{tx{:}transition.\ preconditions(tx)} \tag{11}$$

$$\forall \mathtt{V},\ \textbf{old}(\mathtt{trigger(V)} \land \neg \mathtt{good(V)}) \longrightarrow \mathtt{trigger(V)} \lor \mathtt{good(V)}, \tag{12}$$

then the liveness property in Eq. (8) *holds.*

Proof. Suppose Eq.(8) does not hold. Then either there exists an infinite sequence of transitions starting with a `trigger(V)` state and none of the states has `good(V)`, or there exists a sequence of transitions starting from a `trigger(V)` state and resulting in a dead state where no transitions can be taken before reaching a `good(V)` state. By Eq.(12), a state with `trigger(V)` but not `good(V)` will always be transitioned into a `trigger(V)` ∨ `good(V)` state, which can then be reduced to `trigger(V)` ∧ ¬`good(V)`, since none of the states in either sequence have `good(V)`. Thus, all the states in either sequence must always stay at `trigger(V)` ∧ ¬`good(V)`. By Eq. (11), such states cannot be dead. Thus the sequence must be infinite. We can then construct an integer sequence by calculating the ranking function of each state. By Eq. (9), this sequence is decreasing, and by Eq. (10), this sequence is nonnegative. Thus, we construct an infinite, decreasing sequence of nonnegative integers, which does not exist. This is a contradiction, so Eq.(8) must hold. □

The liveness-to-safety reduction in Theorem 1 works for liveness properties expressible using Eq. (8). Compared with the three properties for the ticket lock protocol in Section 2, Eq. (9) and (10) correspond to the first "termination" property but are defined using a ranking function, Eq. (11) corresponds to the second "no deadlock" property, and Eq. (12) corresponds to the third "ending in good state" property. We refer to the condition `trigger(V)` ∧ ¬`good(V)`, which appears in all Eq. (9-12), as the *liveness prerequisite*. When the liveness prerequisite holds, the desired "good" thing is yet to happen, and the ranking function must decrease and remain nonnegative. There is no constraint on the ranking function when the liveness prerequisite does not hold. For the ticket lock example, the liveness prerequisite is `waiting(C)` ∧ ¬`entered(C)`.

Unlike earlier liveness-to-safety reductions based on acyclicity [Biere et al. 2002; Padon et al. 2017], LVR does not introduce any auxiliary variables or transitions into the protocol. This protocol-preserving character brings two significant benefits. First, it makes LVR easily usable by anyone that understands a distributed protocol, without needing to additionally study and understand some reduction mechanism. Second, it allows LVR to utilize off-the-shelf inference tools to prove safety properties automatically. In contrast, acyclicity-based reduction leads to an augmented protocol that is more complex than the original protocol and beyond the power of automated invariant inference tools, requiring manual proofs for the reduced safety properties [Padon et al. 2017].

Eq. (12) says that the trigger condition keeps holding before entering a "good" state, which is common for many protocols including all of those we evaluated in Section 9. However, LVR can also be used with a protocol whose trigger condition holds for just one moment, without entering a "good" state. To verify such a protocol with LVR, its mypyvy specification can be written to include an auxiliary state `trigger'(V)`. `trigger'(V)` is initialized to false, becomes and remains true when `trigger(V)` holds at just one moment, and becomes false when `good(V)` happens. Then `trigger'(V)` can be used in place of `trigger(V)` when proving Eq. (9-12).

4 INFERENCE OF BOUNDS AND DELTAS

Using ranking functions reduces proving liveness to proving safety which is much easier, but the key challenge is finding such a ranking function. LVR takes a new approach to synthesizing ranking functions by analyzing integer variables of a distributed protocol. It analyzes their upper and lower bounds and deltas. Since the ranking functions usually return integers, it is reasonable to limit the functions we consider to be polynomials of integer variables. We first show how LVR infers the set of integer variables to use when synthesizing a ranking function along with their bounds and deltas, with limited user guidance. We do not attempt completeness throughout our analysis. Instead, we aim to make LVR simple and effective on practical distributed protocols, so it can be used by protocol developers with limited expertise in formal verification.

4.1 Integer Variables

LVR considers three sources of integer variables. The first source is all integer variables directly defined in the protocol specification, including those used in the fairness assumptions. For example, n_{exec} is an integer variable defined in the ticket lock protocol specification.

The second source is the functions of variables that return integers, even if the variables themselves may not be integers, such as `timesched(C)` at Line 6 in Figure 3 for the ticket lock protocol. In general, the functions are directly defined in the protocol specification. However, if an ordering relation is defined in the specification, LVR automatically derives a subtraction function that returns an integer representing the distance between two ordered objects. For example, the ticket lock specification necessarily defines an ordering relation for tickets, so LVR introduces a subtraction function $sub : \texttt{ticket} \times \texttt{ticket} \rightarrow int$ resulting in integer variables such as $sub(\texttt{myt(C)}, \texttt{now})$. LVR enumerates all function-induced integer variables from the protocol specification then removes redundant ones. For example, after $sub(\texttt{myt(C)}, \texttt{now})$ and $sub(\texttt{myt(C)}, \texttt{next})$ are enumerated, $sub(\texttt{now}, \texttt{next})$ is redundant because $sub(\texttt{now}, \texttt{next}) = sub(\texttt{myt(C)}, \texttt{next}) - sub(\texttt{myt(C)}, \texttt{now})$. Any ranking function with $sub(\texttt{now}, \texttt{next})$ can be rewritten using $sub(\texttt{myt(C)}, \texttt{next})$ and $sub(\texttt{myt(C)}, \texttt{now})$.

The third source is integer variables derived from first-order relations defined in the protocol specification. For example, n_{wait} and n_{enter} are derived from first-order relations in ticket lock. There can be many such variables derived from the relations, and it is hard to tell which ones are relevant to liveness reasoning. LVR uses a simple heuristic to identify any variables which are counts corresponding to the first-order relations that appear in the liveness prerequisite or fairness assumptions. For example, the ticket lock protocol has a liveness prerequisite of $\texttt{waiting(C)} \land \neg\texttt{entered(C)}$, so LVR automatically identifies the corresponding counts n_{wait} and n_{enter}.

Besides the automatically inferred variables, an LVR user, such as the protocol developer, can add additional variables based on human intuition. For the ticket lock protocol, the liveness proof intuitively involves the client holding the serving ticket, but the client holding the serving ticket is not explicitly tracked in the protocol specification in Figure 3. Therefore, it makes sense for the user to introduce a variable `active` denoting the client that holds the serving ticket by declaring it via the following proposition: $\texttt{myt(active)} = \texttt{now} \land \neg\texttt{idle(active)}$. With `active` declared, `timesched(active)` is automatically introduced as an integer variable. Section 9 shows that only limited user guidance is generally needed in finding variables. For any variable declared via a proposition, LVR automatically adds two proof targets, namely the existence and uniqueness of the variable under the given liveness prerequisite, to ensure the variable is well-defined. For `active` in ticket lock, the two proof targets are:

$$\forall \texttt{C:client.}\ \texttt{waiting(C)} \land \neg\texttt{entered(C)} \rightarrow \exists \texttt{C': client.}\ \texttt{myt(C')} = \texttt{now} \land \neg\texttt{idle(C')}$$

$$\forall \texttt{C:client.}\ \texttt{waiting(C)} \land \neg\texttt{entered(C)} \rightarrow \forall \texttt{C}_1\texttt{: client, C}_2\texttt{: client.}\ \texttt{myt(C}_1\texttt{)} = \texttt{now} \land \neg\texttt{idle(C}_1\texttt{)} \land \texttt{myt(C}_2\texttt{)} = \texttt{now} \land \neg\texttt{idle(C}_2\texttt{)} \rightarrow \texttt{C}_1 = \texttt{C}_2.$$

The two proof targets are both safety properties and can be proved using an automated invariant inference tool such as P-FOL-IC3.

4.2 Bounds

Given a set of integer variables, LVR determines their coefficients to synthesize a ranking function. Whether a ranking function decreases after a transition eventually comes down to how the mutable integer variables in the ranking function change. A variable is mutable, and declared as such in mypyvy, if it can change during protocol transitions. For example, Figure 3 shows that n_{exec} is mutable but M_{exec} is not. To analyze how mutable integer variables change, LVR first needs to determine the upper and lower bounds of these variables. For example, Table 1 shows the bounds for the mutable integer variables for the ticket lock protocol. Variables n_{exec}, `timesched(C)`, and `timesched(active)` are introduced from fairness assumptions and defined with upper bounds M_{exec} and M_{period}. From the liveness prerequisite `waiting(C)` $\land$ ¬`entered(C)`, the serving ticket `now` has not reached the ticket of a waiting client `C`, so $sub(\texttt{myt(C)}, \texttt{now}) \geq 0$. Since `next` is the next free ticket to allocate, $sub(\texttt{myt(C)}, \texttt{next}) \leq 0$. Since n_{wait} and n_{enter} count subsets of clients, they are bound to be nonnegative. Additionally, the ticket lock protocol ensures mutual exclusion, so n_{enter} cannot exceed 1.

LVR automatically infers these bounds using simple rule-based static analysis, which is formalized in Algorithm 1. LVR first infers conservative variable bounds in the first phase of the algorithm, then further refines the bounds by considering constraints from protocol transitions in the second phase of the algorithm.

In Phase 1, LVR retrieves bounds defined directly from trusted invariants or axioms. For the ticket lock protocol in Figure 3, the trusted invariants at Lines 51 and 52 give upper bounds `timesched(C)` $\leq M_{\text{period}}$, `timesched(active)` $\leq M_{\text{period}}$, and $n_{\text{exec}} \leq M_{\text{exec}}$. LVR then uses information from variable definitions. If the variable is a count over first-order relations, its lower bound is at least zero and its upper bound is constrained by the product of the maximum number of each type of object (*MaxObj*) in the relation; *MaxObj* is ∞ by default. For the ticket lock protocol, the number of waiting clients n_{wait} is defined as `count (C:client) where waiting(C)`, and the number of entered clients n_{enter} is defined as `count (C:client) where entered(C)`, so this count rule gives lower bounds $n_{\text{wait}} \geq 0$ and $n_{\text{enter}} \geq 0$. If the variable is an indicator for some boolean expression, its lower bound is at least zero and its upper bound is at most one. The ticket lock protocol has no indicator variables, but other protocols may use them, such as alternating bit and sliding window, which are discussed in Section 9.

In Phase 2, LVR infers more accurate bounds using the transition information. Phase 2 guarantees that the final bounds define a subrange of the conservative bounds from Phase 1. For each variable v, LVR first initializes its bound according to the protocol initialization, then relaxes the bound for each transition based on how the transition updates the variable. If the transition does not change variable v, the bound remains the same. If the transition sets v to some constant *Const*, the bound is

Algorithm 1 Static Inference of Integer Variable Bounds

```
Input: Distributed protocol $\mathcal{P}$ with transitions $Txs$ and integer variables $Vars$
Output: Lower bound $lower[v]$ and upper bound $upper[v]$ for each $v \in Vars$
1:  // Phase 1: infer conservative "hard" bounds that are independent of transitions
2:  for $v \in Vars$ do
3:      $hardLower[v]$, $hardUpper[v]$ := from_trusted($\mathcal{P}$, $v$)
4:      if $v$ is count $(X_1 : sort_1, ..., X_n : sort_n)$ where $f(X_1, ..., X_n)$ then
5:          $hardLower[v] := \max\{0, hardLower[v]\}$
6:          $hardUpper[v] := \min\{\Pi_{i\in[1,n]} MaxObj(sort_i), hardUpper[v]\}$
7:      else if $v$ is indicator $boolExpr$ then
8:          $hardLower[v] := \max\{0, hardLower[v]\}$
9:          $hardUpper[v] := \min\{1, hardUpper[v]\}$
10: // Phase 2: infer more accurate bounds according to transitions
11: for $v \in Vars$ do
12:     $lower[v]$, $upper[v]$ := from_inits($\mathcal{P}$, $v$)
13:     $lower[v] := \max\{lower[v], hardLower[v]\}$
14:     $upper[v] := \min\{upper[v], hardUpper[v]\}$
15:     for $tx \in Txs$ do
16:         // $new(v)|_{tx}$ denotes the value of $v$ after transition $tx$
17:         if $new(v)|_{tx} = v$ then
18:             pass
19:         else if $new(v)|_{tx} = Const$ then
20:             assert $hardLower[v] \le Const \le hardUpper[v]$
21:             $lower[v] := \min\{lower[v], Const\}$
22:             $upper[v] := \max\{upper[v], Const\}$
23:         else if $new(v)|_{tx} \ge v$ then
24:             $upper[v] := hardUpper[v]$
25:         else if $new(v)|_{tx} \le v$ then
26:             $lower[v] := hardLower[v]$
27:         else
28:             $lower[v] := hardLower[v]$
29:             $upper[v] := hardUpper[v]$
```

relaxed to include *Const*. If the transition increases v, its upper bound is relaxed to the conservative upper bound. If the transition decreases v, its lower bound is relaxed to the conservative lower bound. If the transition can either increase or decrease v, both its lower and upper bounds will be relaxed. For example, for the ticket lock protocol, the conservative bound from Phase 1 for n_{exec} is $(-\infty, M_{\text{exec}}]$, which is too loose. In Phase 2, n_{exec} starts with $[0, 0]$ after initialization, and is changed in only two transitions `execute` and `leave`. `execute` increments n_{exec} and falls into the case at Line 23, so the bound is relaxed to $[0, M_{\text{exec}}]$. `leave` sets n_{exec} to 0 and falls into the case at Line 19, so the bound is relaxed to include 0, which means it remains $[0, M_{\text{exec}}]$.

The rule-based static analysis is incomplete, yet our experience is that it works reasonably well for most protocols. The bounds inferred may be too loose but are never too tight. The user should look through the inferred bounds, and tighten them if needed. For the ticket lock protocol, LVR does not determine the tight upper bound of 1 on n_{enter}. This tight bound is unlikely to be achieved via any static analysis of the protocol specification, even though it is obvious to a human since

Table 1. Bounds of integer variables in the ticket lock protocol.

variable	lower bound	upper bound
n_{exec}	0	M_{exec}
`timesched(C)`	0	M_{period}
`timesched(active)`	0	M_{period}
sub(`myt(C)`, `now`)	0	∞
sub(`myt(C)`, `next`)	$-\infty$	0
n_{wait}	0	∞
n_{enter}	0	1

mutual exclusion means that no more than one client can enter the critical section. This is because there is no mutual exclusion property explicitly in the specification itself. The same applies to sub(`myt(C)`, `now`) ≥ 0, the other tight bound LVR does not infer. LVR will rely on the user to tighten these two bounds.

LVR verifies the bounds are correct using mypyvy. Each bound is a safety property on the protocol (e.g., `timesched(C)` $\geq 0 \land$ `timesched(C)` $\leq M_{\text{period}}$), so verifying it involves finding an inductive invariant. An invariant is inductive if it holds on initial protocol states, is preserved after each transition, and implies the bound. Most bounds are inductive and can be directly verified by mypyvy. For example, all the bounds in Table 1 are inductive except for two, $n_{\text{enter}} \leq 1$ and sub(`myt(C)`, `now`) ≥ 0. For bounds that are not inductive, LVR uses an off-the-shelf invariant inference tool to find an inductive invariant that implies the bound, specifically P-FOL-IC3 because it supports the most recent version of mypyvy.

4.3 Deltas

Once LVR determines the bounds, it needs to analyze further how mutable integer variables in the ranking function change on each transition. These deltas help further constrain the coefficients of the ranking function. Deltas are determined as either actual integer values or in terms of their respective lower and upper bounds.

For each transition, it is important to account for how its arguments can affect those deltas if the arguments overlap with the quantified variables `V` of interest from Theorem 1 or existing variables in the specification. For example, for the ticket lock protocol, we are interested in proving that a client `C` that is waiting will eventually be able to acquire the lock, specified in the liveness prerequisite `waiting(C)` $\land \neg$`entered(C)`. Since each transition has a client `c` as an argument, LVR separately considers deltas for each transition when the argument `c` $=$ `C` or `c` $\neq$ `C`. Since there is a special client `active` that holds the serving ticket, LVR also distinguishes between `c` $=$ `active` or `c` $\neq$ `active`. If a transition has a branch, LVR decomposes the transition into two, one corresponding to the *true* branch and the other for the *false* branch. This allows fine-grained analysis of deltas in each case.

Table 2 shows the deltas for each transition of the ticket lock protocol. For example, for transition `enter(c)`, its precondition `myt(c)` $=$ `now` implies that only client `active` can take this transition, so `c`$\neq$`active` is a contradiction. Therefore, this transition can be ignored for any client `c`$\neq$`active` since it will not affect those deltas. If `c` is our client of interest `C`, four variables will change. n_{wait} decreases by `1`. n_{enter} increases by `1`. Since `C` is scheduled in this transition, `timesched(C)` and `timesched(active)` are reset to `0`. Since `timesched(C)` and `timesched(active)` have lower bound 0 and upper bound M_{period}, their deltas must be in range $[-M_{\text{period}}, 0]$. This also shows why LVR first determines the bounds for each variable, as they are used in the delta analysis. If `c` is not our client

Table 2. Deltas of integer variables after each transition in ticket lock.

transition	variable	delta
`get(c≠C∧c≠active)`	n_{wait}	1
	sub(`myt(C)`, `next`)	$(-\infty$, -1]
	`timesched(C)`	1
	`timesched(active)`	1
`fail(c=C∧c≠active)`	`timesched(C)`	$[-M_{\text{period}}, 0]$
	`timesched(active)`	1
`fail(c≠C∧c≠active)`	`timesched(C)`	1
	`timesched(active)`	1
`enter(c=C∧c=active)`	n_{wait}	-1
	n_{enter}	1
	`timesched(C)`	$[-M_{\text{period}}, 0]$
	`timesched(active)`	$[-M_{\text{period}}, 0]$
`enter(c≠C∧c=active)`	n_{wait}	-1
	n_{enter}	1
	`timesched(C)`	1
	`timesched(active)`	$[-M_{\text{period}}, 0]$
`execute(c≠C∧c=active)`	n_{exec}	1
	`timesched(C)`	1
	`timesched(active)`	$[-M_{\text{period}}, 0]$
`leave(c≠C∧c=active)`	n_{enter}	-1
	n_{exec}	$[-M_{\text{exec}}, 0]$
	sub(`myt(C)`, `now`)	$(-\infty$,-1]
	`timesched(C)`	1
	`timesched(active)`	$[-M_{\text{period}}, 0]$
others	contradiction	contradiction

of interest `C`, `timesched(C)` increases by 1, while the deltas of the other three variables remain the same.

LVR first automatically infers contradictions to eliminate transitions that do not need to be considered in computing deltas. Let L be the liveness prerequisite and P be the precondition of some transition. If $L \longrightarrow \neg P$ holds for the protocol, then the transition is contradictory. For example, `get(c)` in Figure 3 has a precondition P = `idle(c)`, while the liveness prerequisite L = `waiting(C)` ∧ ¬`entered(C)` says that our client of interest `C` is already waiting, so intuitively the transition `get(c)` where `c` = `C` is not allowed. Formally, LVR checks if

$$\forall \texttt{C: client, c: client.}\ \texttt{waiting(C)} \land \neg\texttt{entered(C)} \land \texttt{c} = \texttt{C} \longrightarrow \neg\texttt{idle(c)}$$

holds for the protocol by calling the invariant inference tool P-FOL-IC3, which returns an inductive invariant. This means $L \longrightarrow \neg P$ holds for the protocol so the transition `get(c)` where `c` = `C` is contradictory. If P-FOL-IC3 instead returns `safety violation`, there is no contradiction. Whether each transition is contradictory is independent, so LVR checks their respective $L \longrightarrow \neg P$ in parallel.

For transitions that are not contradictory, LVR then automatically infers deltas using simple rule-based static analysis, which is formalized in Algorithm 2. LVR iteratively checks if the update of variable v matches one of five patterns in Lines 2-16, and calculates the delta accordingly. If none of the patterns is matched, LVR will conservatively assume the variable can change arbitrarily and give the loosest delta $[lower[v] - upper[v], upper[v] - lower[v]]$ at Lines 18-19. Similar to bounds,

Algorithm 2 **Static Inference of Integer Variable Deltas**

Input: Distributed protocol $\mathcal{P}$, transition tx, integer variables *Vars* with their bounds *upper* and *lower*

Output: Upper and lower bound of delta $deltaUpper[v]|_{tx}$ and $deltaLower[v]|_{tx}$ for each $v \in Vars$ on transition tx

```
1:  for $v \in Vars$ do
2:      if $new(v)|_{tx} = v + Const$ then
3:          $deltaLower[v]|_{tx} := Const$
4:          $deltaUpper[v]|_{tx} := Const$
5:      else if $new(v)|_{tx} = v + v'$ then
6:          $deltaLower[v]|_{tx} := lower[v']$
7:          $deltaUpper[v]|_{tx} := upper[v']$
8:      else if $new(v)|_{tx} = Const$ then
9:          $deltaLower[v]|_{tx} := Const - upper[v]$
10:         $deltaUpper[v]|_{tx} := Const - lower[v]$
11:     else if $new(v)|_{tx} \geq v + Const$ then
12:         $deltaLower[v]|_{tx} := Const$
13:         $deltaUpper[v]|_{tx} := upper[v] - lower[v]$
14:     else if $new(v)|_{tx} \leq v + Const$ then
15:         $deltaLower[v]|_{tx} := lower[v] - upper[v]$
16:         $deltaUpper[v]|_{tx} := Const$
17:     else
18:         $deltaLower[v]|_{tx} := lower[v] - upper[v]$
19:         $deltaUpper[v]|_{tx} := upper[v] - lower[v]$
```

the deltas inferred via static analysis can only be too loose, and the user can manually tighten them if needed. For the ticket lock protocol, all deltas are accurately inferred by LVR.

To verify the deltas are correct, LVR proves that the deltas hold in the distributed protocol using mypyvy. To do this, LVR temporarily adds auxiliary variables to the protocol that record old values of integer variables before a transition (e.g., old_n_{wait}). Then the delta of an integer variable is encoded as a safety property (e.g., $n_{wait} = old\_n_{wait} + 1$ for transition `get(c ≠ C ∧ c ≠ active)`). LVR verifies deltas for different transitions in parallel. Bounds and contradictions that have been verified earlier are encoded as trusted invariants, as the inductiveness of the delta properties is dependent on them.

4.4 Protocol Refinement for Automatically Proving Safety Properties

LVR is designed to leverage invariant inference tools such as P-FOL-IC3, but even the ticket lock protocol specification in Figure 3 has integer variables. Unfortunately, invariant inference tools currently can only find inductive invariants to prove safety properties in first-order logic. They do not find invariants with integer arithmetic [Goel and Sakallah 2021a; Hance et al. 2021; Karbyshev et al. 2017; Ma et al. 2019; Padon et al. 2022]. To address this problem, a key observation is that the state transitions generally only involve first-order relations while integer variables are used just to encode fairness assumptions. For example, the ticket lock protocol only uses integer variables n_{exec}, `timesched`, M_{exec}, and M_{period} for its fairness assumptions in Figure 3. We refer to the variables used for encoding fairness assumptions as fairness variables, the protocol with fairness variables as the *fairness-included protocol*, and the protocol with all fairness variables removed as the *fairness-free protocol*.

To use an invariant inference tool, LVR automatically removes all fairness variables, including all initializations, preconditions, and assignments that include them, to transform the *fairness-included protocol* to its *fairness-free protocol*. LVR also automatically transforms the safety property when needed. For example, the bound $n_{\text{enter}} \leq 1$ for the ticket lock protocol is translated into:

$$\forall \mathtt{C}_1\mathtt{: client, C}_2\mathtt{: client.\,entered(C}_1\mathtt{)} \land \mathtt{entered(C}_2\mathtt{)} \rightarrow \mathtt{C}_1 = \mathtt{C}_2.$$

We can then use the fairness-free protocol for proving safety properties based on:

Theorem 2. *For any fairness-included protocol $\mathcal{P}_I$ that never assigns a fairness variable to a non-fairness variable, let $\mathcal{P}_F$ be its induced fairness-free protocol. For any safety property $\mathcal{S}$, if $\mathcal{S}$ holds on $\mathcal{P}_F$, then $\mathcal{S}$ must also hold on $\mathcal{P}_I$.*

Proof. Let Σ_I and Σ_F be the protocol state space for $\mathcal{P}_I$ and $\mathcal{P}_F$. Let $f : \Sigma_I \longrightarrow \Sigma_F$ be the projection function from a fairness-included protocol state to a fairness-free protocol state by removing all fairness variables. For any $\sigma_I \in \Sigma_I$, if σ_I satisfies the protocol initialization conditions of $\mathcal{P}_I$, $f(\sigma_I)$ is guaranteed to satisfy the initialization conditions of $\mathcal{P}_F$, because the initialization conditions for $\mathcal{P}_F$ is a subset of those for $\mathcal{P}_I$.

Next consider any transition $\sigma_{I_1} \xrightarrow{tx} \sigma_{I_2}$ in $\mathcal{P}_I$. The transition tx is represented by a transition name with parameters. Since the preconditions of tx in $\mathcal{P}_F$ are a subset of those in $\mathcal{P}_I$, tx must also satisfy its preconditions for $f(\sigma_{I_1})$ in $\mathcal{P}_F$. Since there is no assignment from fairness variables to non-fairness variables, $f(\sigma_{I_1}) \xrightarrow{tx} f(\sigma_{I_2})$ is a valid transition in $\mathcal{P}_F$.

The two facts above establish a refinement relation between state machines of $\mathcal{P}_I$ and $\mathcal{P}_F$. For any state $\sigma_I^* \in \Sigma_I$ reachable via $\sigma_{I_0} \rightsquigarrow \sigma_I^*$ in $\mathcal{P}_I$, we know $f(\sigma_I^*)$ is reachable via $f(\sigma_{I_0}) \rightsquigarrow f(\sigma_I^*)$ in $\mathcal{P}_F$. So any safety property $\mathcal{S}$ that holds on $\mathcal{P}_F$ also holds on $\mathcal{P}_I$. □

Theorem 2 allows us to prove safety properties on the fairness-free protocol, where invariant inference tools are more likely to be applicable, then automatically extend the result to the fairness-included protocol. Section 9 shows that all the protocols we evaluated except for one have fairness-free counterparts that contain no arithmetic so that an invariant inference tool can use them to verify bounds and deltas. The only exception is the sliding window protocol, where integer arithmetic is embedded in the protocol itself to reason about the window size `width`. As a result, the user has to manually write an inductive invariant to verify bounds and contradictions for this protocol, as shown in Table 5.

Theorem 2 does not require fairness variables to be completely independent of protocol variables, although they usually are in practice. Instead, it only requires that the fairness variables are never assigned to protocol variables. If some precondition involves both fairness and protocol variables, it will be removed in the fairness-free protocol. This may lead to weaker preconditions in the fairness-free protocol, and thus additional safety violations. Consequently, Theorem 2 does not guarantee that a safety violation in the fairness-free protocol implies a safety violation in the fairness-included protocol, so theoretically, the invariant inference tool may fail on the fairness-free protocol when an inductive invariant does exist for the fairness-included protocol. However, LVR does not expect the invariant inference tool to always succeed in the first place. When the inference tool times out or throws an error, the problem is simply deferred to the user to manually check the property and write an inductive invariant if needed.

The user does not need to understand the state-machine refinement in Theorem 2, since both the protocol retrofitting and the property translation are handled by LVR behind the scenes. The user is only involved when the protocol does not satisfy the prerequisite of Theorem 2. Specifically, LVR will alert the user if a fairness variable is assigned to a non-fairness variable, which indicates that the fairness assumption may have been written incorrectly.

After P-FOL-IC3 returns an inductive invariant for some safety property, LVR feeds that inductive invariant to the fairness-included protocol and lets mypyvy validate it. In this way, the user does not need to trust LVR's retrofitting and translation procedure.

5 SMT-BASED RANKING FUNCTION SYNTHESIS

Based on the proven bounds and deltas, LVR assembles a ranking function using the integer variables. LVR formulates the ranking function as a weighted linear combination of mutable integer variables; Section 6 discusses more complex functions. Inferring the ranking function reduces to determining how much weight to give to each variable, which can be expressed as a set of coefficients that can be solved by an SMT solver.

For example, for the ticket lock protocol, the ranking function has the shape:

$$\begin{aligned} R(\mathtt{C}) = {} & W_1 \times sub(\mathtt{myt(C)}, \mathtt{now}) + W_2 \times sub(\mathtt{myt(C)}, \mathtt{next}) \\ & + W_3 \times n_{\text{exec}} + W_4 \times \mathtt{timesched(C)} \\ & + W_5 \times \mathtt{timesched(active)} + W_6 \times n_{\text{wait}} + W_7 \times n_{\text{enter}}, \end{aligned} \tag{13}$$

where $W_1, ...W_7$ are the weights. Each weight is further formulated as a sub-formula of immutable integer variables. If we allow up to second-degree polynomials of immutable integer variables, then a sub-formula for weight could be

$$\begin{aligned} W_1 = {} & x_1 \times M_{\text{period}}^2 + x_2 \times M_{\text{period}} \times M_{\text{exec}} + x_3 \times M_{\text{exec}}^2 \\ & + x_4 \times M_{\text{period}} + x_5 \times M_{\text{exec}} + x_6. \end{aligned} \tag{14}$$

For the ticket lock protocol, there are two immutable integer variables, so considering all possible combinations of those variables up to a second-degree term results in six terms with six coefficients $x_1, ..., x_6$. Expanding $W_2, ..., W_6$ in the same way defines coefficients $x_7, ..., x_{42}$, resulting in 42 coefficients.

The bounds and deltas on the mutable integer variables, together with the requirement that the ranking function decreases for each transition while remaining nonnegative, will impose a set of constraints on the weights and consequently the coefficients of the immutable integer variables as well. The constraint generation process is formalized in Algorithm 3.

LVR first considers nonnegativity of the ranking function which depends on variables' bounds. If a variable v has an open upper bound, then it cannot have a negative weight in the ranking function (Line 5). If a variable v has an open lower bound, then it cannot have a positive weight in the ranking function (Line 7).

Next for each transition tx, LVR considers the decreasing of the ranking function which depends on variables' deltas. This is a linear programming problem, where the delta of each $v \in Vars$ is a variable x_v with constraint $deltaLower[v]|_{tx} \leq x_v \leq deltaUpper[v]|_{tx}$, and the ranking function changes by $\sum_{v \in Vars} W_v \cdot x_v$ after a transition. The constraints define a feasible region of possible solutions, and max $\sum_{v \in Vars} W_v \cdot x_v$ must occur at a vertex of the feasible region if it exists. Since max $\sum_{v \in Vars} W_v \cdot x_v < 0$ must hold for the ranking function to decrease, LVR adds constraints on the weights to ensure a maximum ranking function change exists and it is less than zero. If a variable v can increase infinitely in a transition, then x_v has an open upper bound and cannot have a positive weight (Line 14). Similarly, if a variable v can decrease infinitely in a transition, then x_v has an open lower bound and cannot have a negative weight (Line 18). A variable x_v has specific values at each vertex, so LVR then enumerates all vertices and adds a constraint on the weights for each vertex so that the ranking function change is negative.

Algorithm 3 **Constraint Generation for Ranking Function**

Input: Integer variables *Vars*, with their bounds *upper* and *lower*, and their deltas $deltaUpper|_{tx}$ and $deltaLower|_{tx}$ for each transition $tx \in Txs$

Output: A set of linear inequality constraints $\mathcal{I}$ over weights

```
 1: $\mathcal{I} := \emptyset$
 2: for $v \in Vars$ do
 3:     declare weight $W_v$
 4:     if $upper[v]$ is $\infty$ then
 5:         $\mathcal{I} := \mathcal{I} \cup \{W_v \geq 0\}$
 6:     if $lower[v]$ is $-\infty$ then
 7:         $\mathcal{I} := \mathcal{I} \cup \{W_v \leq 0\}$
 8: for $tx \in Txs$ do
 9:     $varList := []$
10:     $deltaSetList := []$
11:     for $v \in Vars$ do
12:         $deltaSet := \emptyset$
13:         if $deltaUpper[v]|_{tx}$ is $\infty$ then
14:             $\mathcal{I} := \mathcal{I} \cup \{W_v \leq 0\}$
15:         else
16:             $deltaSet := deltaSet \cup \{deltaUpper[v]|_{tx}\}$
17:         if $deltaLower[v]|_{tx}$ is $-\infty$ then
18:             $\mathcal{I} := \mathcal{I} \cup \{W_v \geq 0\}$
19:         else
20:             $deltaSet := deltaSet \cup \{deltaLower[v]|_{tx}\}$
21:         if $deltaSet \neq \emptyset$ then
22:             Append($varList$, $v$)
23:             Append($deltaSetList$, $deltaSet$)
24:     for $deltaEachVar \in$ CartesianProduct($deltaSetList[0], ..., deltaSetList[|varList|-1]$) do
25:         $\mathcal{I} := \mathcal{I} \cup \{\sum_{i \in [0,|varList|-1]} deltaEachVar[i] \cdot W_{varList[i]} < 0\}$
```

For simplicity, suppose no variable $v \in Vars$ can increase or decrease infinitely, and for each $v \in Vars$ we have $deltaUpper[v]|_{tx} \neq deltaLower[v]|_{tx}$. Then the feasible region is a $|Vars|$-dimensional hyperrectangle. For each $v \in Vars$, *deltaSet* will have two elements $deltaUpper[v]|_{tx}$ and $deltaLower[v]|_{tx}$, added at Lines 16 and 20. Then the Cartesian product over the $|Vars|$ deltaSets will give $2^{|Vars|}$ elements, each representing a vertex of the hyperrectangle (Line 24). For each vertex, LVR adds a constraint saying that the change of the ranking function is negative (Line 25). In practice, the number of generated constraints is usually much fewer than $2^{|Vars|}$, since a transition usually changes only a subset of integer variables, and among them many have their deltas as a number (i.e., $deltaUpper[v]|_{tx} = deltaLower[v]|_{tx}$) instead of a range.

Now we have a set of constraints over weights $W_1, W_2, ...$ and consequently the underlying coefficients $x_1, x_2, ...$. Solving for the coefficients across a set of inequalities can be extremely complicated, especially for multivariable and higher-order functions. We make two observations regarding practical distributed protocols that can simplify this analysis. First, we observe that the immutable variables are generally nonnegative integers, such as the one used for bounds in the case of the ticket lock protocol. Second, we conjecture that there should exist a simple ranking function.

Based on these observations, LVR makes a simplifying assumption that for any inequality, the weight and the coefficients of all of its monomials have the same sign. For example, if $W_1 \geq 0$, then all $x_1, ...x_{10} \geq 0$. Similarly if $W_1 \leq 0$, then all $x_1, ...x_{10} \leq 0$. By assuming the weights and coefficients have the same sign, complex minima/maxima analysis of multivariate high-order functions can be reduced to simple linear inequalities. For example, suppose $W_1 = x_1 \times M_{\text{exec}}^2 + x_2 \times M_{\text{exec}} + x3$, and some constraint requires $W_1 \geq 0$ for any $M_{\text{exec}} \geq 0$. This constraint can be satisfied if $(x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_3 \geq 0)$ or $(x_1 > 0 \wedge x_2 < 0 \wedge 4x_1x_3 - x_2^2 \geq 0)$. LVR only considers the former and not the latter, simplifying its analysis.

In other words, the constraints on all of the coefficients can be expressed as simple linear inequalities. For example, $sub(\texttt{myt(C)}, \texttt{now})$ has an open upper bound ∞, so $W_1 \geq 0$ otherwise the ranking function $R(\texttt{C})$ cannot be nonnegative. For transition `execute(c≠C∧c=active)`, the delta in Table 2 shows that n_{exec} and `timesched(C)` increases by 1, and `timesched(active)` changes by $[-M_{\text{period}}, 0]$. To make the ranking function decrease after this transition, LVR gets $W_3 + W_4 < 0$ and $W_3 + W_4 - M_{\text{period}} \times W_5 < 0$. These constraints are then decomposed into inequalities of coefficients $x_1, ..., x_{42}$.

Eventually 94 linear inequalities are generated. Any assignment of the 42 coefficients that satisfies the 94 inequalities corresponds to a valid ranking function. LVR feeds the inequalities to the Z3 SMT solver, which returns a solution $x_2 = 1, x_4 = 2, x_5 = 1, x_6 = 2, x_{16} = -1, x_{18} = -1, x_{30} = -1, x_{40} = -1, x_{42} = -1$, with every other x_i being 0. Then LVR automatically adds terms involving only immutable integer variables to make the ranking function nonnegative, leading to the final ranking function Eq. (7) for the ticket lock protocol.

While the ranking function shape in Eq. (13) is linear with regard to mutable integer variables, LVR can also infer piecewise linear ranking functions. In the TLB shootdown protocol discussed in Section 9, the ranking function is piecewise linear and depends on the program counter pc:

$$R(\texttt{C}) = \begin{cases} R_1(\texttt{C}) & pc = i1 \\ R_2(\texttt{C}) & pc = i2 \\ \dots \end{cases}$$

The translation to inequalities is mostly the same as above, with additional considerations to switch the ranking function. While ranking functions can theoretically be of any form, we observe that simpler linear and piecewise linear ranking functions of mutable integer variables can often be sufficient for practical distributed protocols, especially in combination with tiering as discussed in Section 6.

Validation of ranking functions. Once the ranking function has been synthesized, LVR verifies it using mypyvy. Since the synthesis of the ranking function was itself accomplished using an SMT solver to verify different steps of the synthesis process, it is expected that the ranking function will be correct. However, we perform an additional validation step to ensure that the correctness of the ranking function does not need to trust LVR and is guarded by an SMT solver. LVR validates the ranking function by providing mypyvy with the prerequisite, protocol specification, bounds and deltas, inductive invariants for verifying bounds and deltas, as well as the ranking function itself. mypyvy feeds all the information into the underlying Z3 SMT solver to verify the ranking function is strictly decreasing and nonnegative for each protocol transition. Alternatively, if the bounds and deltas are trusted since they were previously verified using an SMT solver, they do not have to be reverified and the inductive invariants for verifying the bounds and deltas can be omitted in verifying the ranking function, reducing the constraints that need to be considered by the SMT solver. Since whether a ranking function decreases and remains nonnegative after one

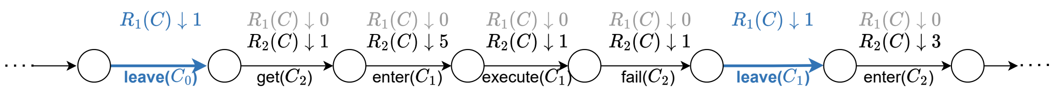


Fig. 5. Tiered ranking function for ticket lock protocol. Both R_1(C) and R_2(C) are nonnegative. The top-tier ranking function R_1(C) strictly decreases on transition leave and does not increase on others. So the system will terminate within finite steps of leave. The second tier ranking function R_2(C) strictly decreases on every transition other than leave, so within finite steps of these transitions, the system will either take transition leave or terminate. Thus, the system will terminate in finite steps, regardless of what transitions are taken.

transition is independent of another transition, LVR validates the ranking function on different transitions in parallel.

6 TIERED RANKING FUNCTIONS FOR SCALABILITY

6.1 Motivation and Example

Although the ranking functions previously discussed are linear combinations of mutable integer variables, they are higher-degree polynomials from the perspective of an SMT solver in terms of reasoning about a combination of both mutable and immutable integer variables. For example, the ranking function for ticket lock protocol in Eq. (7) is a third-degree polynomial. For more complex protocols, the degree can be higher with many more terms, which poses challenges to not only synthesizing the ranking function but also validating it. For example, for the sliding window network transmission protocol discussed in Section 9, the entire ranking function is actually a fourth-degree polynomial, on which mypyvy times out after 24 hours when trying to prove, all at once, that it decreases after a transition.

To address this problem, LVR introduces tiered ranking functions to modularize liveness reasoning and make proofs easier for SMT solvers. Rather than synthesizing a ranking function all at once, which is valid for all protocol transitions, we can decompose the transitions into tiers and synthesize a ranking function for each tier, then verify ranking functions incrementally tier by tier. Each tier encompasses a subset of the transitions. LVR synthesizes a tiered ranking function for the first tier, ignoring any transitions not included in that tier. The tiered ranking function is the same as a regular ranking function, except it only has to be strictly decreasing for transitions in the respective tier. For transitions not included in that tier but part of subsequent tiers, the tiered ranking function can remain unchanged, but it cannot increase. Once a tiered ranking function is found, that intuitively means that the liveness property can be verified for the distributed protocol assuming only transitions in that tier occur.

However, we also need to make sure that transitions occurring in other tiers do not repeatedly happen between transitions included in the first tier such that they prevent first-tier transitions from happening. LVR accomplishes this by next synthesizing a tiered ranking function for the next tier. Once that tiered ranking function is found, that intuitively means that those transitions will not go on forever and prevent the first-tier transitions from occurring. This implies that the liveness property can be verified for the distributed protocol assuming only transitions in the first two tiers occur. The same process is repeated for each tier to verify the liveness property across all tiers, and therefore all transitions.

For example, we can use tiered ranking functions to prove the liveness of the ticket lock protocol, instead of constructing one ranking function all at once for all transitions. Figure 5 gives an illustration. At a high level, a waiting client C eventually will enter the critical section because the serving ticket increases whenever a client leaves the critical section, so it will sooner or later reach client `C`'s ticket so it can enter the critical section. This can be stated as "if transition `leave` keeps happening, then our desired `entered(C)` will happen." If `leave` is the only transition included in a first tier, its liveness property can be proved using a tiered ranking function that decreases in transition `leave` and does not increase in other transitions while remaining nonnegative. Such a ranking function can be automatically inferred in the same way as discussed in Section 5. The only difference is that, for transitions other than `leave`, the ranking function can remain unchanged (but not increase), so < 0 will be replaced by ≤ 0 in the respective constraints. LVR successfully finds a first-tier ranking function

$$R_1(\texttt{C}) = sub(\texttt{myt(C)}, \texttt{now}).$$

The liveness property will hold as long as no other transitions keep `leave` from happening. If all other transitions are included in a second tier, then we want to prove that those transitions do not happen infinitely often to keep `leave` from happening, or the `enter` transition in the second tier directly occurs for `C`, so the desired `entered(C)` condition happens. In other words, we want to prove a liveness property for the remaining transitions, not including `leave`. This can be done using a tiered ranking function that decreases in all transitions except `leave` while remaining nonnegative. This can be automatically inferred as well in the same way as discussed in Section 5. LVR successfully finds a second-tier ranking function

$$\begin{aligned} R_2(\texttt{C}) = (M_{\text{period}} + 1) \times (M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}}) \\ + M_{\text{period}} - \texttt{timesched(active)}. \end{aligned}$$

In this way, the ranking function in Eq. (7), a third-degree polynomial with 14 terms, is decomposed into two smaller tiered ones of lower degree, the first having only one term, and the second being a second-degree polynomial with eight terms. Each tiered ranking function can be separately verified by an SMT solver, avoiding the need for the solver to consider higher-degree polynomials.

6.2 General Form and Proof of Correctness

More formally, a k-tier ranking function is a length-k sequence of pairs:

$$[(R_1(V), TX_1); ...; (R_k(V), TX_k)],$$

where $R_i(V)$ is the ranking function for the i-th tier ($i \in [1, ..., k]$), and TX_i is the set of transitions belonging to that tier. The union $\bigcup_{1 \leq i \leq k} TX_i$ must be the set of all transitions in the protocol.

For tiered ranking functions, it is necessary to prove that it decreases after each transition and remains nonnegative, similar to single ranking functions, but the proof needs to be done for each tier. Specifically, we need to prove nonnegativity for each tier:

$$\forall i \in [1, ..., k],\ \forall \texttt{V}.\ (\texttt{trigger(V)} \land \neg\texttt{good(V)}) \longrightarrow R_i(\texttt{V}) \geq 0. \tag{15}$$

We also need to prove it decreases for each tier such that after any transition in tier i, the ranking function of tier i strictly decreases, as well as all higher-tier ranking functions do not increase:

$$\begin{aligned} \forall i \in [1, ..., k], \forall tx \in TX_i,\ \forall \texttt{V}.\ \textbf{old}(\texttt{trigger(V)} \land \neg\texttt{good(V)}) \longrightarrow \\ R_i(\texttt{V}) < \textbf{old}(R_i(\texttt{V})) \land \\ \textstyle\bigwedge_{j \in [1,...,i-1]} R_j(\texttt{V}) \leq \textbf{old}(R_j(\texttt{V})). \end{aligned} \tag{16}$$

The soundness of tiered ranking functions for proving liveness properties is established by:

Theorem 3. *If there exists a k-tier ranking function* $[(R_1(V), TX_1); ...; (R_k(V), TX_k)]$ *that satisfies Eq.* (15)(16)*, then the protocol cannot have an infinite sequence of states satisfying* `trigger(V)` $\land$ `¬good(V)`.

Proof. We prove this by induction. For the base case when $k = 1$, the ranking function has only one tier, and Eq. (15)(16) degenerate to Eq. (9)(10) in Section 3. Then the proof for Theorem 1 directly applies.

Now suppose the proposition holds for any n-tier ranking function. We will show it also holds any $(n+1)$-tier ranking function by contradiction. Suppose we have an infinite sequence of states satisfying `trigger(V)` $\land \neg$`good(V)`. Let $i = 1$ in Eq. (15)(16). We get

$$\forall \mathtt{V}.\ (\mathtt{trigger(V)} \land \neg \mathtt{good(V)}) \longrightarrow R_1(\mathtt{V}) \geq 0 \tag{17}$$

$$\forall tx \in TX_1,\ \forall \mathtt{V}.\ \mathbf{old}(\mathtt{trigger(V)} \land \neg \mathtt{good(V)}) \longrightarrow R_1(\mathtt{V}) < \mathbf{old}(R_1(\mathtt{V})). \tag{18}$$

Let σ_1 be a state in the infinite sequence. From Eq. (17) we know $R_1(V)$ is a nonnegative integer at σ_1. Eq. (18) tells that $R_1(V)$ strictly decreases whenever a transition $tx \in TX_1$ is taken. So the number of transitions belonging to TX_1 after σ_1 must be finite. Let σ_2 be the state after the last transition $tx \in TX_1$ in the infinite sequence. The sequence after σ_2 is still an infinite sequence satisfying `trigger(V)` $\land \neg$`good(V)` but does not involve TX_1. We can construct a new protocol without TX_1 and a new n-tier ranking function $[(R_2(V), TX_2); ...; (R_k(V), TX_k)]$. The new ranking function satisfies Eq. (15)(16) for the new protocol. From the induction hypothesis, it does not include any infinite sequence of states satisfying `trigger(V)` $\land \neg$`good(V)`, a contradiction. □

The proof of Theorem 3 establishes the logical equivalence between a k-tier ranking function and a lexicographic ranking function of length k. The difference is that the former can be easily encoded and reasoned about in mypyvy, while the latter cannot be directly expressed in mypyvy due to its restrictions to first-order logic and integer arithmetic.

To use tiered ranking functions to prove liveness for a distributed protocol, LVR requires the user to identify which transitions belong to each tier and the number of tiers to consider. Tiering is not unique, so users can make different choices of tiers when verifying the liveness properties of a protocol. Our evaluation suggests that tiering is not required to verify liveness properties for most distributed protocols, but can be useful in some cases and requires only a modest amount of additional information from the user.

7 THEORETICAL GAP IN ABSTRACT BOUND

When encoding the fairness assumptions of ticket lock in Section 2, we let variable `timesched`(C) track the time since a client `C` was last scheduled, and give an abstract declaration of M_{period} representing the upper bound of the time delta from one scheduled to the next. The fairness assumption that every client is scheduled infinitely often is then transformed to Eq. (6) which is

$$\forall \mathtt{C:\ client},\ \forall \mathtt{T:\ time},\ \mathtt{timesched(C)}|_{\mathtt{T}} \leq M_{\text{period}}.$$

Based on this fairness assumption, LVR proves liveness for *any* M_{period} instead of some specific value. We mentioned in Section 2 that there is a minor theoretical difference between Eq. (5) and Eq. (6). Specifically, there exist execution traces in which client `C` is scheduled infinitely often, but the scheduling interval has no upper bound—if a client is scheduled at time $1, 2, 4, ..., 2^n, ...$ and the sequence grows infinitely, then temporal logic formulation in Eq. (5) holds, but no M_{period} exists that satisfies Eq. (6).

This does not mean the liveness proof is unsound. Instead, it relies on a slightly stronger fairness assumption. If some server's implementation satisfies the fairness assumption in Eq. (5) but not Eq. (6), then the liveness proof does not apply to such a server implementation.

LVR can also prove liveness without introducing abstract bounds. If one prefers to verify ticket lock using the original formulation in Eq. (5), instead of a global bound M_{period}, LVR will introduce a prophecy variable `period(C)` that predicts when client `C` will be scheduled next. `period(C)` is nondeterministically reset to any nonnegative integer when `C` is scheduled. The fairness assumption will then be encoded as

$$\forall \texttt{C: client}, \forall \texttt{T: time}, \texttt{timesched(C)}|_{\texttt{T}} \leq \texttt{period(C)}.$$

In other words, `period(C)` is a dynamic bound for each scheduling decision. However, without an upper bound on `period(C)`, when `C` is scheduled `period(C)` will change by $(-\infty, \infty)$. The simplest ranking function will be the three-tuple

$$(\mathit{sub}(\texttt{myt(C)}, \texttt{now}),\ M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}},\ \texttt{period(active)} - \texttt{timesched(active)})$$

in lexicographic order. This can also be written as a three-tier ranking function. $\mathit{sub}(\texttt{myt(C)}, \texttt{now})$ strictly decreases on transition `leave`. $M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}}$ strictly decreases on transitions `enter` and `execute`. `period(active)` − `timesched(active)` strictly decreases on transitions `get` and `fail`. As long as the user puts the five transitions in the aforementioned three tiers, LVR can automatically infer the ranking function of each tier.

From our perspective, the small theoretical gain of prophecy variables is outweighed by the additional complexity and human effort in tiering, so abstract bounds are used by default. A user can make their own choice when using LVR.

8 PROVING ABSENCE OF DEADLOCKS

Under the liveness verification scheme formalized in Theorem 1, besides proving a ranking function, we need to prove two safety properties: the system has no deadlock (Eq. (11)), and can only end in a good state (Eq. (12)). The second property is usually easy to prove. For ticket lock, this means a waiting client cannot go back to idle without entering the critical section. The property is inductive and can be directly proved by mypyvy.

To prove the first "no deadlock" property, we need to show that at any state before the desired liveness condition is satisfied, there must be at least one transition that can be taken. A transition is takeable if its preconditions are satisfied. For example, for the ticket lock protocol, this means

$$\begin{aligned} \forall \texttt{C: client.}\ \texttt{waiting(C)} \land \neg\texttt{entered(C)} \longrightarrow \\ (\exists \texttt{C': client.}\ \text{preconditions of } \texttt{get(C')}) \\ \lor (\exists \texttt{C': client.}\ \text{preconditions of } \texttt{fail(C')}) \\ \lor (\exists \texttt{C': client.}\ \text{preconditions of } \texttt{enter(C')}) \\ \lor (\exists \texttt{C': client.}\ \text{preconditions of } \texttt{execute(C')}) \\ \lor (\exists \texttt{C': client.}\ \text{preconditions of } \texttt{leave(C')}). \end{aligned} \tag{19}$$

In other words, at least one of the five transitions must be takeable for some client `C`. This is a safety property, so we can prove it by using an off-the-shelf tool to find the inductive invariant, same as proving bounds and deltas. P-FOL-IC3 successfully finds an inductive invariant to prove Eq. (19) in two minutes.

For some complex protocols, examining all transitions to prove the absence of deadlock may result in too many terms to consider to find the inductive invariant, causing the underlying SMT solver used by an invariant inference tool to time out. For example, if a distributed protocol has 10 transitions, each transition has three preconditions, and each precondition has two terms, there are 60 terms to resolve in a single formula, which is beyond the capabilities of existing automated invariant inference tools. To avoid this, LVR allows the user to provide additional guidance to simplify invariant inference by specifying a subset of transitions that are sufficient to ensure the absence of deadlock and eliminating existentially quantified variables.

For example, for the ticket lock protocol, we can simplify Eq. (19) by specifying which transitions need to be considered. We know at any time before the desired `entered(C)` condition is satisfied, 1) if no client is holding the critical section, then entering is a valid transition for the client whose local ticket equal to `now`, or 2) if some client `c'` is holding the critical section, then leaving is a valid transition for `c'`. We can therefore narrow the set of transitions to consider from five to two, namely `enter` and `leave`, while excluding the other three transitions `get`, `fail`, and `execute`. If we further expand the exact preconditions, we can simplify Eq. (19) to:

$$\begin{aligned}\forall \texttt{C: client. waiting(C)} \land \neg\texttt{entered(C)} \longrightarrow \\ (\exists \texttt{C': client. waiting(C')} \land \texttt{myt(C')} = \texttt{now}) \\ \lor (\exists \texttt{C': client. entered(C')}). \end{aligned} \tag{20}$$

We can further eliminate an existential quantifier because we know which `C'` can enter or leave. "The client whose local ticket equals to now" is exactly the declared variable `active`, whose existence and uniqueness have been verified earlier (see Section 4.1), so one can narrow the choice of `C'` to exactly `active`. This further simplifies Eq. (20) to:

$$\begin{aligned}\forall \texttt{C: client. waiting(C)} \land \neg\texttt{entered(C)} \longrightarrow \\ (\texttt{waiting(active)} \land \texttt{myt(active)} = \texttt{now}) \\ \lor (\texttt{entered(active)}). \end{aligned} \tag{21}$$

Note that we only use the ticket lock protocol as an example of how user guidance can simplify invariant inference for proving deadlock freedom. In fact, the ticket lock protocol is simple enough that user guidance is actually not necessary to prevent the underlying SMT solver from timing out when proving the absence of deadlocks.

9 EVALUATION

9.1 Distributed Protocols and Their Liveness

We demonstrate the effectiveness of LVR by evaluating it on eight distributed protocols, including all protocols whose liveness has been manually verified by either Ironfleet [Hawblitzel et al. 2015a] or Dynamic Abstraction [Padon et al. 2017, 2021].

Table 3 describes each protocol and its liveness property. Each liveness property also requires fairness assumptions. Ticket lock requires fair scheduling and finite execution in the critical section. Hybrid reliable broadcast (HRB), alternating bit, and sliding window are network protocols that require some minimum guarantee on packet delivery. TLB shootdown requires scheduling and lock fairness. Paxos and its variants require some period of synchrony with some minimum guarantee on packet delivery, and limited contention between proposers across rounds. Specifically, Paxos requires that there exists a round r_0 which no later round contends with. The leader of round r_0 carries out its duty in initializing the round and proposing a value. For any packet that belongs to round r_0, if it is sent between the leader and a participant that belongs to a live quorum q_0, then it will eventually be delivered.

For protocols manually verified by Dynamic Abstraction, we translate their Ivy specifications [Padon et al. 2017, 2021] to its twin language, mypyvy. We use mypyvy because it is written in Python 3 and supports both Z3 and CVC4 SMT solvers. We observe that for some protocols, the same ranking function can be validated by mypyvy but times out on Ivy. For the sliding window protocol which was manually verified by IronFleet, we reimplemented its Dafny specifications [Hawblitzel et al. 2015b] in mypyvy, which involved some necessary rewriting due to language differences. The reimplementation is logically equivalent but not a line-by-line translation. IronFleet's sliding window protocol is used as a module to guarantee packet delivery between hosts in a key-value

Table 3. Distributed protocols evaluated.

name	**description**
ticket lock	Shown in Figure 3. Liveness property: any waiting client will eventually enter the critical section.
HRB	Broadcasts messages in a network consisting of correct, omittable, and byzantine nodes. Liveness property: any correct node will eventually accept a message broadcast by all non-byzantine nodes.
alternating bit [Tel 2000]	Involves a sender and receiver transmitting over a channel with packet loss. Liveness property: any data from the sender will eventually be received by the receiver.
TLB shootdown [Hoenicke et al. 2017]	Ensures TLB consistency. Liveness property: after a shared memory write, each processor will eventually update its TLB correctly.
Paxos [Lamport 1998, 2001]	Enables nodes to reach a consensus value in an unreliable network. Liveness property: eventually some consensus must be reached.
multi-Paxos [Lamport 2001]	An extension of Paxos, where nodes reach consensus on a growing log instead of a single value.
stoppable Paxos [Malkhi et al. 2008]	An extension of multi-Paxos that allows reconfiguration of the system (e.g., adding new nodes).
sliding window [Tel 2000]	An extension of alternating bit where the sender can send consecutive pieces of data up to its window size, and the network can both drop and reorder packets.

store, but how each host maintains its hashtable and generates new data to send is unrelated to the protocol. For IronFleet's sliding window, our reimplementation implements the network protocol. The rest of the system is unrelated to its liveness reasoning, so we abstract the host by simply introducing a `gen_data` transition that nondeterministically generates a new message to be sent, similar to the alternating bit protocol in Dynamic Abstraction. Algebraic datatypes in Dafny are encoded as relations or functions in mypyvy.

9.2 Results and Discussion

Tables 4 and 5 show the results for using LVR to verify the liveness for each protocol. In all cases, LVR automatically synthesizes a ranking function for each protocol and verifies all eight protocols, while only requiring limited user guidance. Table 4 shows various protocol characteristics, and characteristics of the verification process for each protocol. It shows the number of transitions and lines of code (LoC). It also shows the number of mutable integer variables from three sources: explicitly declared as `int` in the specification, generated from `int`-typed functions, and derived from first-order relations.

Table 5 shows the runtime for each stage in LVR: 1) static analysis to determine integer variables, bounds, and deltas, 2) proving bounds, 3) determining which transitions are contradictory, 4) proving deltas for non-contradictory transitions, 5) synthesizing the ranking function with the Z3 SMT solver, 6) validating the ranking function using mypyvy, 7) proving the "ending in good state" property, and 8) proving the absence of deadlock. For steps 2, 3, 4, 7, and 8, P-FOL-IC3 is called when the property is not inductive. If P-FOL-IC3 times out determining whether a transition is contradictory, the transition is conservatively considered non-contradictory; a one hour timeout threshold is used. The user can add and prove a missing contradiction by rerunning P-FOL-IC3 or manually writing the inductive invariant. In our experiments, LVR keeps running until reaching

Table 4. Protocol statistics and user guidance for verifying liveness properties of distributed protocols.

			integer variables			user guidance				
protocol name	transitions	LoC	declare	function	derive	variable	bound	contradiction	delta	tier
ticket lock	5	68	1	4	2	1	2	0	0	-
HRB	8	137	0	0	2	2	0	0	0	-
alternating bit	9	127	3	2	5	3	2	0	0	2
TLB shootdown	27	361	0	0	25	1	0	3	0	4
Paxos	9	133	0	0	9	4	0	3	0	-
multi-Paxos	10	135	0	0	10	5	0	0	0	-
stoppable Paxos	11	160	0	0	10	5	0	0	0	-
sliding window	8	119	1	6	2	1	2	2	0	4

Table 5. Runtime breakdown for verifying liveness properties of distributed protocols. Runtime is measured in seconds unless otherwise specified.

protocol name	static	bound	contradiction	delta	synthesis	validate	good-end	no-deadlock
ticket lock	0.02	48.3	127.5	0.57^{s}	0.21	0.56	0.43^{s}	117.4
HRB	0.06	0.64^{s}	1.07^{n}	0.65^{s}	0.19	0.64	0.59^{s}	86.2
alternating bit	0.05	560.8	1.46^{n}	5.11	0.39	1.68	0.78^{s}	0.83^{s}
TLB shootdown	0.43	10.3^{s}	$6h^{m}$	24.0	1.75	17.3	0^{d}	872.7
Paxos	0.11	0.91^{s}	4929	0.67^{s}	0.20	0.71	0^{d}	533.3
multi-Paxos	0.13	1.02^{s}	544.1	0.68^{s}	0.21	0.71	0^{d}	287.8
stoppable Paxos	0.18	1.29^{s}	3113	0.79^{s}	0.23	0.82	0^{d}	1979.8
sliding window	0.14	$2.5h^{m}$	$1.5h^{m}$	12.6	1.54	4.55	0.57^{s}	0.59^{s}

s = Property is inductive by itself. Runtime is just for mypyvy to check it.
m = Inductive invariant found manually because P-FOL-IC3 does not support or times out. Time reported in hours.
n = No transition is contradictory. Runtime reported is just for preprocessing.
d = Degenerate case. Liveness property has trigger condition *true*, so "ending in good state" property of Eq. (12) becomes a tautology.

the timeout threshold, but Table 5 shows the actual runtime required until the last contradiction is found. When tiered ranking functions are used, the results reported are the sum totals across all tiers. In the LVR workflow, static analysis, constraint solving, and ranking function validation are all deterministic and the runtime is stable across trials, while invariant inference using P-FOL-IC3 has highly variable runtimes due to “variations in seeds and the non-determinism of parallelism” [Koenig et al. 2022]. For each P-FOL-IC3 query, we run it three times and report the average.

Table 4 shows user guidance needed for synthesizing ranking functions in terms of the number of: additional variables introduced, bounds adjusted, contradictions added, deltas adjusted, and tiers provided as input if tiered ranking functions were used. To provide guidance, users only need an understanding of the distributed protocol, not the internal workings of LVR or SMT solvers. Given an understanding of the protocol, providing hints is straightforward in all cases. Each protocol requires some user guidance in adding variables. The number of integer variables listed for each protocol in Table 4 includes those that require user guidance. In some cases, the explicit user guidance provided may not be the actual integer variable. For example, as discussed in Section 4.1, for the ticket lock protocol, an `active` variable is introduced via user guidance, from which `timesched(active)` is automatically introduced as an integer variable based on an integer function. Other than the ticket lock protocol, user guidance for all other protocols is required only for derived variables.

Most protocols also require some hints from the user to tighten bounds or deltas or specify tiers. For example, Table 4 shows that Paxos requires three corrections on deltas, all in the form of adding missing contradictions. One such contradiction is about transition `cast_vote`, which lets a node n receive a previously unseen proposed value v at round r, and cast a vote for it under some condition. P-FOL-IC3 does not recognize the transition is contradictory under condition $\exists V' : \texttt{value}.\ \texttt{proposal\_received}(n, r, V')$, but a human developer with knowledge of the Paxos protocol can quickly tell that this cannot happen — there cannot be two conflicting proposed values v and V' in one round, because each round has only one proposer who can only propose one value. The other two missing contradictions are of similar nature. If the user identifies one, it is easy to correct all. If the user does not provide enough variables or tighten all necessary bounds or deltas, Z3 will output "no solution" when solving coefficients, indicating that something is missing to the user. For all protocols, all user guidance for deltas involved adding contradictions, rather than adjusting the upper or lower bound of deltas for a non-contradictory transition.

For all protocols, user guidance is not necessary to prove the absence of deadlocks. However, if desired, user guidance by excluding transitions or eliminating existential quantifiers can reduce the runtimes for protocols whose no-deadlock safety property is not itself already inductive. For example, including user guidance as discussed in Section 8, the runtime to prove the absence of deadlocks would be cut in half to only a minute for ticket lock and could be reduced by more than a factor of three for stoppable Paxos.

Table 5 shows that LVR reduces liveness to safety properties that can mostly be automatically verified. As discussed in Section 3, the liveness property is reduced to two safety properties— Eq. (12) (ending in good state) and Eq. (11) (no deadlock)—and a ranking function property. The ranking function property is decomposed into three more safety properties—proving bounds, proving contradictions, and proving deltas, before synthesizing the polynomial function. Therefore, there are a total of five safety properties per protocol, or 40 total for eight protocols. A dominant majority of safety properties (37 out of 40) are either always true, inductive by themselves thus automatically verified by mypyvy, or have an inductive invariant found by P-FOL-IC3 automatically. Only three safety properties could not be verified by P-FOL-IC3, so those inductive invariants were found manually in a few hours each.

Table 5 shows that the time to synthesize the ranking function, after verifying bounds and deltas, was a tiny fraction of the overall runtime. Ranking function synthesis just requires solving coefficients for a set of linear inequalities, which is where SMT solvers excel. Every Z3 query to synthesize a ranking function finished in less than a second. For sliding window and TLB shootdown, a ranking function is synthesized for each tier, so their total runtime slightly exceeds one second. In most cases, Z3 synthesizes the exact same ranking function as a human would manually specify. Overall, the results and runtimes show that LVR was successful in significantly reducing the human effort to verify the liveness properties.

IronFleet previously manually verified liveness for two of the eight protocols, sliding window in IronKV and multi-Paxos in IronRSL, which required 2,093 and 7,869 LoC in Dafny, respectively. In comparison, using LVR, even if we include all lines in the protocol specification, then count each user-specified or corrected variable, bound, delta, and tier as one LoC, the manual effort to verify sliding window and multi-Paxos are 128 and 165 LoC, reducing the coding effort by 16 and 47 times, respectively. While part of the difference is due to Dafny versus mypyvy and the protocol encoding (IronFleet's multi-Paxos includes the local scheduler and IO queue used in its low-level implementation while our multi-Paxos is more abstracted), LVR still saves significant manual effort, with most reasoning and proofs automated using off-the-shelf tools and SMT solvers.

Dynamic Abstraction previously manually verified seven of the eight protocols, but required a large number of additional auxiliary relations and transitions to augment the protocols. For example, for Paxos, the user needs to introduce an additional 25 relations and 143 LoC in the augmented protocol, resulting in more than three times as many relations and more than twice the code as the original protocol. More importantly, Dynamic Abstraction requires 40 invariants for the augmented protocol to verify the liveness of Paxos. The sheer size of the augmented protocols and invariants required is beyond the reach of any existing invariant inference methods, so Dynamic Abstraction requires significant human effort to manually verify the safety property of the augmented protocol. In contrast, LVR leverages automated invariant inference and SMT solvers to verify liveness for Paxos with limited user guidance to add variables and adjust deltas. This is possible because its ranking function approach involves much fewer and shorter invariants, and it avoids requiring a more complex augmented protocol with many more relations.

Limitations. LVR only targets liveness properties that say something eventually happens (see Eq. (8)), not that something happens within or after a specific time. For example, byzantine consensus protocols for blockchains may have liveness properties specifying that consensus is reached within a certain time [Chan and Shi 2020]. LVR only synthesizes ranking functions as polynomials over integer variables, or combinations of such polynomials in lexicographic order, while theoretically, a ranking function can have any shape. There also exist polynomial ranking functions that cannot be synthesized from variables' deltas. For example, if a transition tx nondeterministically resets variables x and y ($x, y \in \mathbb{N}$), while only guaranteeing that $new(x)|_{tx} + new(y)|_{tx} = x + y - 1$, then $x + y$ is a valid ranking function, although the deltas of both x and y are $(-\infty, \infty)$ from which no ranking function can be synthesized, unless the user explicitly specifies $x + y$ as an auxiliary variable. LVR inherits the limitations on expressiveness and theorem solving capabilities of mypyvy and the underlying Z3 and CVC4 solvers. As discussed in Section 6, SMT solvers may time out when reasoning about high-order polynomials, in which case LVR will not be able to validate the ranking function.

10 RELATED WORK

Most work on verifying distributed protocols and systems only prove safety properties [Hance et al. 2020; Wilcox et al. 2015; Woos et al. 2016]. Since the introduction of Ivy [Padon et al. 2016] and its successor mypyvy [Wilcox et al. 2018], a growing number of methods have been proposed to automatically infer inductive invariants to prove safety properties of distributed protocols, including I4 [Ma et al. 2019], SWISS [Hance et al. 2021], DuoAI [Yao et al. 2022], pdH [Padon et al. 2022], P-FOL-IC3 [Koenig et al. 2022], among others [Goel and Sakallah 2021a,b; Karbyshev et al. 2017; Koenig et al. 2020; Yao et al. 2021]. None of the aforementioned systems verify any liveness properties or provide guidance on extending the systems to accommodate liveness properties. Unlike many of these tools [Hance et al. 2021; Karbyshev et al. 2017; Ma et al. 2019; Padon et al. 2022; Yao et al. 2022, 2021], LVR is not confined to effectively propositional logic (EPR).

IronFleet [Hawblitzel et al. 2015a] enables manually verifying distributed systems by proving refinement from the implementation to some state-machine model then to a centralized specification. It encodes and reasons about liveness properties and fairness assumptions in temporal logic, but verifying liveness involves writing thousands of lines of Dafny and the painstaking process of tuning triggers to tame the SMT solver. To verify the liveness of a new protocol, a user has to either tune triggers themselves, or study and stick to the predefined TLA proof rules in IronFleet's library, both requiring expertise in Dafny and SMT solvers. In contrast, LVR introduces a liveness-to-safety reduction using ranking-function-based analysis. This not only eliminates the need for temporal logic reasoning, but also unleashes the power of automated invariant inference to significantly reduce human effort.

Biere et al. [2002] introduces the classical liveness-to-safety reduction for finite-state systems, in which liveness is established by the acyclicity of system states in any execution. Padon et al. [2017, 2021] proposes *Dynamic Abstraction* to generalize this reduction to infinite-state systems, which most distributed protocols can be written as. It reduces the liveness property to a fair cycle detection problem, which is representable as a safety property in an augmented protocol with many additional variables and transitions. The safety property can be proved with an inductive invariant, but the high complexity of the augmented protocol renders automated invariant inference tools useless. Manually writing invariants requires a thorough understanding of Dynamic Abstraction. LVR reduces liveness to safety, but on the original protocol using ranking functions, which is compatible with automated tools and is much easier to understand for users.

Ranking functions, also called measure functions [Kaufmann et al. 2013], have long been used to prove the termination of loops and recursions in programs [Cook et al. 2007a]. There is extensive work on automatically synthesizing linear [Colón and Sipma 2002; Gonnord et al. 2015; Heizmann and Leike 2015; Podelski and Rybalchenko 2004], polynomial [Neumann et al. 2020], lexicographic [Cook et al. 2013], and multi-phase [Ben-Amram and Genaim 2017] ranking functions. There are also heuristic approaches to proving fair termination for multithreaded programs without explicit ranking functions [Cook et al. 2007b; Farzan et al. 2016; Kragl et al. 2020]. However, these methods target only arithmetic, executable programs. The formulation and reasoning of distributed protocols usually involves richer structure, in particular the universal and/or existential quantification over nodes, messages, etc., in first-order logic, which is beyond the scope of previous work. LVR allows both first-order relations and nonlinear integer arithmetic in a protocol, although as discussed in Section 4.4, using integers in fairness assumptions only can best utilize the power of currently available automated invariant inference tools.

There is also a line of work studying liveness verification for multithreaded programs and general infinite-state systems, proving theoretical results on decidability under various settings [Atig et al. 2012; Baumann et al. 2021; Fortin et al. 2017]. LVR does not aim to give new results on decidability. Instead, it provides theorems and a new framework to enable the mostly automated verification of liveness properties for distributed protocols.

11 CONCLUSIONS

LVR is the first system to verify the liveness properties of distributed protocols in a mostly automated manner. LVR reduces the liveness proof to synthesizing a ranking function and finding inductive invariants that imply safety properties. We show for the first time that ranking functions for common distributed protocols can be automatically synthesized as polynomial formulas of integer protocol variables. To accomplish this, LVR analyzes the bounds of the variables and how much they change across protocol transitions, proving them as safety properties, then uses these constraints to solve for coefficients to synthesize a formula that is monotonically decreasing and nonnegative, which can be efficiently done using an SMT solver. Using ranking functions allows LVR to prove liveness by leveraging existing off-the-shelf invariant inference and invariant checking tools to find inductive invariants and verify safety properties. LVR marks a significant step forward in the state of the art, requiring significantly less human effort and expertise to verify liveness properties; users do not need to understand the internal workings of SMT solvers or other tools. Our results demonstrate LVR's effectiveness in automatically verifying liveness for eight distributed protocols for the first time, including three versions of Paxos, with only limited user guidance.

DATA AVAILABILITY STATEMENT

The artifact of this paper [Yao et al. 2023], including the source code of LVR and evaluated protocols, is available on Zenodo.

ACKNOWLEDGMENTS

Carla Ferreira provided helpful comments on earlier drafts. This work was supported in part by an Amazon Research Award, a VMware Systems Research Award, an NSF CAREER Award, DARPA contract N66001-21-C-4018, and NSF grants CCF-2124080, CCF-2239484, CNS-2052947, and CNS-2247370. Ronghui Gu is a co-founder of and has an equity interest in CertiK.

REFERENCES

Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. 2012. Detecting fair non-termination in multithreaded programs. In *Proceedings of 24th International Conference on Computer Aided Verification (CAV '12)*. 210–226. https://doi.org/10.1007/978-3-642-31424-7_19

Pascal Baumann, Rupak Majumdar, Ramanathan S Thinniyam, and Georg Zetzsche. 2021. Context-bounded verification of liveness properties for multithreaded shared-memory programs. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–31. https://doi.org/10.1145/3434325

Amir M Ben-Amram and Samir Genaim. 2017. On multiphase-linear ranking functions. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV '17)*. 601–620. https://doi.org/10.1007/978-3-319-63390-9_32

Armin Biere, Cyrille Artho, and Viktor Schuppan. 2002. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science* 66, 2 (2002), 160–177. https://doi.org/10.1016/S1571-0661(04)80410-9

Benjamin Y. Chan and Elaine Shi. 2020. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies (AFT '20)*. 1–11. https://doi.org/10.1145/3419614.3423256

Michael A. Colón and Henny B. Sipma. 2002. Practical methods for proving program termination. In *Proceedings of 14th International Conference on Computer Aided Verification (CAV '02)*. 442–454. https://doi.org/10.1007/3-540-45657-0_36

Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007a. Proving that programs eventually do something good. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. 265–276. https://doi.org/10.1145/1190216.1190257

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2007b. Proving thread termination. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 320–330. https://doi.org/10.1145/1273442.1250771

Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. lexicographic termination proving. In *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*. 47–61. https://doi.org/10.1007/978-3-642-36742-7_4

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving liveness of parameterized programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. 185–196. https://doi.org/10.1145/2933575.2935310

Marie Fortin, Anca Muscholl, and Igor Walukiewicz. 2017. Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In *Proceedings of 29th International Conference on Computer Aided Verification (CAV '17)*. 155–175. https://doi.org/10.1007/978-3-319-63390-9_9

Aman Goel and Karem Sakallah. 2021a. On symmetry and quantification: A new approach to verify distributed protocols. In *Proceedings of the 13th NASA Formal Methods Symposium (NFM '21)*. 131–150. https://doi.org/10.1007/978-3-030-76384-8_9

Aman Goel and Karem A Sakallah. 2021b. Towards an automatic proof of Lamport's Paxos. In *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*. 112–122. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_20

Laure Gonnord, David Monniaux, and Gabriel Radanne. 2015. Synthesis of ranking functions using extremal counterexamples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 608–618. https://doi.org/10.1145/2737924.2737976

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding invariants of distributed systems: It's a small (enough) world after all. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 115–131.

Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 99–115.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015a. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015b. The IronFleet repository. https://github.com/microsoft/Ironclad/tree/main/ironfleet.

Matthias Heizmann and Jan Leike. 2015. Ranking templates for linear loops. *Logical Methods in Computer Science* 11 (2015). https://doi.org/10.2168/LMCS-11(1:16)2015

Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. 2017. Thread modularity at many levels: A pearl in compositional verification. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. 473–485. https://doi.org/10.1145/3009837.3009893

Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2017. Property-directed inference of universal invariants or proving their absence. *J. ACM* 64, 1, Article 7 (2017), 33 pages. https://doi.org/10.1145/3022187

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 2013. *Computer-aided reasoning: ACL2 case studies*. Vol. 4. Springer Science & Business Media. https://doi.org/10.1007/978-1-4757-3188-0

Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. 703–717. https://doi.org/10.1145/3385412.3386018

Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. 2022. Inferring invariants with quantifier alternations: Taming the search space explosion. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*. 338–356. https://doi.org/10.1007/978-3-030-99524-9_18

Bernhard Kragl, Constantin Enea, Thomas A Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. 227–242. https://doi.org/10.1145/3385412.3385980

Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

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

Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. 2022. Sift: Using refinement-guided automation to verify complex distributed systems. In *2022 USENIX Annual Technical Conference (USENIX ATC '22)*. 151–166.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. 2019. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 370–384. https://doi.org/10.1145/3341301.3359651

Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. 2008. *Stoppable Paxos*. Technical Report MSR-TR-2008-192. https://www.microsoft.com/en-us/research/publication/stoppable-paxos/

Eike Neumann, Joël Ouaknine, and James Worrell. 2020. On ranking function synthesis and termination for polynomial programs. In *Proceedings of the 31st International Conference on Concurrency Theory (CONCUR '20)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Oded Padon. 2021. Source file of the ticket lock protocol in Ivy. https://github.com/kenmcmil/ivy/blob/master/examples/liveness/ticket_nested.ivy.

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017. Reducing liveness to safety in first-order logic. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33. https://doi.org/10.1145/3158114

Oded Padon, Jochen Hoenicke, Kenneth L McMillan, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2021. Temporal prophecy for proving temporal properties of infinite-state systems. *Formal Methods in System Design* 57, 2 (2021), 246–269. https://doi.org/10.1007/s10703-021-00377-1

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 '16)*. 614–630. https://doi.org/10.1145/2908080.2908118

Oded Padon, James R Wilcox, Jason R Koenig, Kenneth L McMillan, and Alex Aiken. 2022. Induction duality: Primal-dual search for invariants. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 50 (Jan. 2022), 29 pages. https://doi.org/10.1145/3498712

Andreas Podelski and Andrey Rybalchenko. 2004. A complete method for the synthesis of linear ranking functions. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*. 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

Kerry Raymond. 1989. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)* 7, 1 (1989), 61–77. https://doi.org/10.1145/58564.59295

Gerard Tel. 2000. *Introduction to distributed algorithms*. Cambridge university press. https://doi.org/10.1017/CBO9781139168724

James Wilcox, Oded Padon, Yotam Feldman, and other contributors. 2018. The mypyvy language. https://github.com/wilcoxjay/mypyvy.

James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas 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 (PLDI '15)*. 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CCP '16)*. 154–165. https://doi.org/10.1145/2854065.2854081

Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. 485–501.

Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2023. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions (Artifact). https://doi.org/10.5281/zenodo.10039066.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-driven automated invariant learning for distributed protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. 405–421.

Received 2023-07-11; accepted 2023-11-07