Toward Liveness Proofs at Scale

Kenneth L. McMillan(✉)

University of Texas at Austin, Austin, USA
kenmcmil@gmail.com

Abstract. While the problem of mechanized proof of liveness of reactive programs has been studied for decades, there is currently no method of proving liveness that is conceptually simple to apply in practice to realistic problems, can be scaled to large problems without modular decomposition, and does not fail unpredictably due to the use of fragile heuristics. We introduce a method of liveness proof by relational rankings, implement it, and show that it meets these criteria in a realistic industrial case study involving a model of the memory subsystem in a CPU.

1 Introduction

The problem of mechanized proof of liveness of reactive programs has been studied for decades. Yet proving liveness of practical systems remains a challenge that is typically beyond the capability or time constraints of practicing engineers. This is not to say that we lack the conceptual framework or the tools needed to prove liveness properties. Rather, the difficulty lies in applying the tools and methods at the scale and complexity of systems encountered in industry. Here, we study the source of these difficulties with the goal of developing an approach that allows engineers with a reasonable degree of sophistication in formal methods to prove liveness of real systems.

The inspiration to study this problem comes from an effort to prove liveness of models of memory systems that have been developed by hardware engineers at Apple, Inc. The engineers use a tool and language called Ivy [18] to prove safety properties of memory subsystem models. These properties guarantee the consistency of memory operations from the point of view of the processor cores. Liveness of these models is considered important, in part to ensure the liveness of the underlying hardware implementation, but also to guarantee that the consistency proofs are not vacuous, owing to oversights in the models that might result in deadlock. Unfortunately, proving liveness using an existing approach implemented in Ivy was found by the engineers to be excessively difficult.

To understand why this is this is case, we must consider first the safety proofs. These are accomplished using a large number of hand-written inductive invariants of the models that were verified using a decision procedure (the SMT solver Z3 [5]). It is significant that it was possible to construct these proofs and maintain them through model changes on the time scale of industrial processor design.

One important factor in this success is that the models and proofs are constructed in a way that allows automated verification using only *effectively propositional reasoning*, or EPR [22], in which category we include extended logical fragments implemented

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14681, pp. 255–276, 2024.
https://doi.org/10.1007/978-3-031-65627-9_13

in modern SMT solvers, such as FAU [9]. This means that the verifier is a decision procedure and hence can provide both proofs and counterexamples. Empirically, automated verification within EPR is far more efficient, stable and reliable than verification in the richer logics that SMT solvers provide [21,23]. This makes it possible to rapidly iterate while crafting a proof by invariant, while rarely having to debug failures of the verifier. Using EPR also makes it practical to verify a large model representing the entire memory system without resorting to compositional methods (*i.e.*, without using assume/guarantee specifications of the system components). The ability to carry out proof without modular decomposition significantly reduces the conceptual complexity of the proof task for engineers.

Unfortunately, these advantages of EPR do not currently carry over to liveness proofs. Known methods for liveness proof either *(1)* do not produce verification conditions in EPR, *(2)* cannot be applied at the necessary scale or *(3)* are too conceptually complex to be applied in industry. We will enumerate here the primary approaches to liveness proof the and difficulty of applying them in industrial applications.

Model Checking. Because the models are complex and not finite-state it is a significant challenge to apply model checking to them. Model checkers for infinite-state systems generally do not support liveness proofs and do not, in any event, scale to the needed size and complexity even for safety. This rules out model checking approaches such as [8,10,11,13,17] as well as Invisible Ranking [6]. Proofs combining model checking with compositional refinement and abstraction methods are possible [16] but have high conceptual complexity.

Deductive Approaches. In addition to inductive invariants, the common deductive approaches to liveness require the user to provide a *ranking* that maps the system state into a well-founded pre-order [14,15]. This is conceptually simple, but reasoning about well-founded orders generally takes us outside of EPR. We will consider why this is the case in more detail shortly, and how undecidable reasoning makes the problem of constructing large proofs substantially more difficult.

Liveness-to-Safety Translations. In this approach, we prove liveness by proving safety of a transformed program [2]. Although the approach has primarily been applied to finite-state systems, a recent method due to Padon *et al.* can be applied to infinite-state systems and produces verification conditions in EPR [19,20]. Unfortunately, it requires a user to provide an inductive invariant for the transformed program. The conceptual complexity of this task is high due to the subtlety and complexity of the transformation. This is the approach that the Apple engineers found difficult to apply.

Motivated by the practical experience at Apple, the goal of the proposed work is to create a liveness proof methodology with the following characteristics:

1. It is conceptually simple and easy to apply in common cases,
2. It supports a rich class of systems and temporal specifications,
3. It requires users to reason only about their own system, not about automatically constructed state machines,

4. It is as automated as is practical, relying only on decidable automated reasoning, and
5. It can be scaled to large models without modular decomposition.

To achieve these goals, we propose to apply the novel concept of a *relational ranking*. Using a relational ranking, we hypothesize that we can retain the relative simplicity of proof by well-founded ranking, allowing us to express rankings of height up to ω^n while avoiding any automated reasoning about well-founded orders, thus keeping the proof obligations within EPR. This allows us to reason automatically about the entire model, without modular decompositions, and thus without the need to write complete interface specifications for the system components (a task which is difficult for engineers). In place of this, we can re-use the inductive invariants needed to prove safety to supply almost all of the liveness proof. In fact, our experience with a generic memory system model provided by Apple bears out these expectations.

The primary contributions of this work are, first *a novel deductive proof approach for liveness*, based on relational rankings, and second, a *case study of industrial interest*, applying the method and motivating the features of the proof approach.

2 Background and Related Work

The classical approach to proving liveness properties deductively is to apply a *ranking*. A ranking is a function of the system state that ranges over a well-founded pre-order (often the natural numbers). This approach is well known for proving termination of sequential programs and was adapted to the proof temporal logic properties of concurrent, reactive programs by Manna and Pnueli [15]. To apply their ranking rule, the user must supply inductive invariants and a ranking that are expressed over the system state. The proof is then reduced to non-temporal *verification conditions* that in principle can be discharged automatically. Approaches to proving liveness or termination using rankings include [4,12,24], some of which can synthesize rankings in limited cases.

2.1 Liveness-to-Safety with Rankings

We will use linear temporal logic (LTL) to define temporal properties, with $\Box$ for 'always' and $\Diamond$ for 'eventually'. Following Manna and Pnueli, $\rightarrow$ stands for implication, while $\Rightarrow$ stands for temporal entailment. That is, $p \Rightarrow q$ is a shorthand for $\Box(p \rightarrow q)$. We prove properties of standard first-order symbolic transition systems. As usual, we assume a vocabulary Σ of function and relation symbols representing the program state and a corresponding vocabulary of primed symbols Σ' representing the next state. A transition system is a pair $\langle \mathcal{I}, \mathcal{T} \rangle$, where $\mathcal{I}$ is a first-order formula over Σ representing the initial states and $\mathcal{T}$ is a first-order formula over Σ and Σ' representing the set of system transitions. We take $\mathcal{I} \wedge \Box\mathcal{T}$ as an axiom. That is, $\mathcal{I}$ holds initially, and $\mathcal{T}$ for every successive pair of states.

Manna and Pnueli gave a basic rule for proving properties of the form:

$$(\Box\Diamond r) \rightarrow (p \Rightarrow \Diamond q). \tag{1}$$

The condition r is assumed to occur infinitely often. We will call this a *justice constraint*, and the formula r a *justice condition*. If justice condition r holds infinitely

```
init:                    action send(x):          action poll:
  pend := λx. false;       requires x > last;       if ∃t. pend(t):
  last := 0;               last := x;                 x := min_y(pend(y));
                           pend(x) := true;           emit recv(x);
                                                      pend(x) := false;
```

Fig. 1. Simple timestamped queue example in a notional synchronous language.

often, then whenever p holds, q must hold in the future. For now, we will assume that p, r and q are non-temporal. To prove such a formula, we show that p establishes an invariant ϕ that holds until q is true. Moreover, while ϕ holds, a ranking δ never increases, and whenever r holds, δ decreases. We require that δ be a function from the program state into some well-founded set.

In our notation, the rule for proving formulas of form (1) is as follows:

$$\frac{\begin{array}{l}\text{B1. } p \Rightarrow (q \vee \phi)\\ \text{B2. } \phi \Rightarrow (q' \vee (\phi' \wedge (\delta' \leq \delta)))\\ \text{B3. } \phi \wedge r \Rightarrow (q' \vee (\delta' < \delta))\end{array}}{(\Box\Diamond r) \rightarrow (p \Rightarrow \Diamond q)} \tag{2}$$

Notice that premises B1–B3 are temporal entailments. Normally, these are proved using the safety rule, stated below:

$$\frac{\begin{array}{l}\text{I1. } \mathcal{I} \rightarrow \rho\\ \text{I2. } \mathcal{T} \wedge \rho \rightarrow \rho'\\ \text{I3. } \mathcal{T} \wedge \rho \rightarrow p\end{array}}{\Box\, p} \tag{3}$$

Thus, we can think of (2) as a liveness-to-safety rule. We will call the invariant ρ used in the safety rule the 'safety invariant' to distinguish it from the invariant ϕ in the liveness rule.

A Simple Example. As an example, Fig. 1 shows a simple transition system representing a message queue, inspired by the Apple generic memory model. The actions represent atomic system transitions, which we assume are proved terminating (for example, because they are loop-free, as here). A sender enters messages in the queue with logical time stamps, drawn from the natural numbers. We assume the time stamps of messages entering the queue are increasing, but there may be gaps in the time stamp sequence. A receiver polls the queue for messages. When there is a message, the message with minimum time stamp is removed and a signal 'recv' is emitted. The state predicate 'pend' represents the time stamps that are currently present in the queue.

We would like to prove the following property for all t:

$$(\Box\Diamond \text{poll}) \rightarrow (\text{send}(t) \Rightarrow \Diamond \text{recv}(t)). \tag{4}$$

where send(t) is true when sender enters a message with time stamp t, recv(t) is true when the receiver removes a message with time stamp t, and poll is true when the

receiver polls the queue. In other words, if the receiver polls the queue infinitely often, then every sent message is eventually received. Note that t is a temporal constant.

An obvious ranking function for this proof counts the number of time stamps $\leq t$ that are pending in the queue, that is, $\delta = |\{\tau \mid \tau \leq t \wedge \text{pend}(\tau)\}|$. While timestamp t is pending, δ will decrease each time the queue is polled, since the removed time stamp, being minimal, must be $\leq t$. We can define δ in first-order logic using a simple recursion over the natural numbers. That is, $\text{cnt}(\tau)$, the number of pending timestamps $\leq \tau$, is defined recursively as:

$$\text{cnt}(\tau) \triangleq \text{ite}(\tau = 0, 0, \text{cnt}(\tau - 1)) + \text{ite}(\text{pend}(\tau), 1, 0)$$

We then have $\delta = \text{cnt}(t)$. For the liveness invariant ϕ in the proof rule, we can use just $\text{pend}(t)$. Our justice condition r is that the queue is polled, *i.e.*, $r = \text{poll}$.

In principle, we can now discharge the premises of Rule (2) using the safety rule, with safety invariant $\rho = \forall x.\ \text{pend}(x) \rightarrow x \leq \text{last}$. Notice this invariant is also need to prove the safety property that timestamps are dequeued in increasing order. Since all of the premises of the safety rule are (quantified) first-order formulas over the theory of linear arithmetic, we should be able to discharge them automatically using Z3, a powerful automated theorem prover that supports this theory. When we attempt this, however, we obtain a disappointing result. When trying to discharge premise B3 (stating that δ decreases when the queue is polled) Z3 runs for a few minutes and then returns an inconclusive result – neither a proof nor a counterexample. The solvers CVC4 and CVC5 also fail.

The problem is that the formula we want to prove is outside Z3's decidable fragment. To show that removing a timestamp $\leq t$ reduces $\text{cnt}(t)$ requires induction over t. Z3 is unable to do this. The needed inductive generalization may seem trivial in this small example, so one might easily imagine that a heuristic approach could solve the problem. In a large proof, however, such heuristics are fragile and fail in opaque ways. This failure puts a heavy burden on the user to debug the prover heuristics or discover the necessary instance of the induction axiom by hand. One may imagine alternative ranking schemes, for example using list or finite set datatypes to represent the ranking. This does not escape the fundamental problem, however, that reasoning about well-founded sets requires instantiating an induction schema.

To see how these issues play out in a typical ranking approach, consider the ranking-based liveness proof method of [24]. This method synthesizes a ranking as a polynomial over the integer variables in the program as well as the cardinalities of certain predicates appearing in the program. This heuristic is fragile, however. In our simple example, the predicate whose cardinality we require is $\lambda\tau.\ \text{pend}(\tau) \wedge \tau \leq t$. This predicate appears nowhere in the program or the property, and so the ranking synthesis must fail (we cannot build a ranking from just the 'pend' predicate). Moreover, if we use the cardinality of this predicate in the ranking, the method will be unsound, since this cardinality is not verified to be finite. Even given the predicate and a proof of its finiteness, the approach must infer 'deltas' which are upper and lower bounds on the changes in the integer variables for each action. This is done by an approximate analysis. Unfortunately, the method cannot infer an upper bound of -1 for the delta of our predicate for the 'poll' action, because the inference is based on pattern matching and requires

updates to the predicate to occur in the program. Thus the method cannot infer that the ranking is eventually reduced. Even if this were somehow fixed, verifying the resulting safety properties could still fail because the mixing of quantifiers, uninterpreted predicates and non-linear integer arithmetic makes the verification conditions undecidable. The same issues would be faced using other methods based on linear ranking functions over integers (*e.g.*, [4]). The fundamental problem is that unreliable heuristics must be used to skirt undecidability, leading to methods that fail even on very simple problems. The Ironfleet approach [12] is substantially more manual, but still relies on undecidable reasoning about well-founded orders. Users must therefore diagnose opaque and unpredictable failures of the prover and provide manual guidance. By contrast, Invisible Ranking [6,7] produces decidable verification conditions, but cannot express rankings of height above ω.

2.2 Dynamic Liveness-to-Safety Construction

An alternative to well-founded rankings for infinite-state systems is the dynamic liveness-to-safety method (DL2S) proposed by Padon *et al.* [19]. It works by translating a program P and a liveness property ϕ into a *different* program P' and *safety* property ϕ', such that $P' \models \phi'$ implies $P \models \phi$. Space prohibits a detailed description of the method here. At a high level, it is similar to the method of Biere for finite-state systems [2] in that it detects bad cycles by storing a shadow copy of the system state and testing whether the system can return to the shadow state after certain fairness constraints have been satisfied. The DL2S method is similar to the finite-state method, except that it stores in the shadow state only a finite amount of information about the system state that is dynamically chosen. Restricting the shadow state to a finite projection of the system state guarantees that every infinite behavior must visit some shadow state infinitely often.

The DL2S approach is quite flexible, and has the advantage that in common cases, it yields safety verification conditions in EPR. Thus, the automated part of the proof tends to be reliable and we do not obtain inconclusive results as we did above, using a well-founded ranking. Unfortunately, the safety proof requires us to construct an inductive invariant over the transformed program P'. This greatly increases the conceptual complexity of proof development task for the user. Quoting Padon *et* al. [20] "Because our approach relies on an inductive invariant supplied by the user, it requires the user to understand the liveness-to-safety transformation and it requires both cleverness and a deep understanding of the protocol." To illustrate this, Fig. 2 shows an invariant obtained manually for the simple queue example of Fig. 1, after many failed attempts and counterexamples. The symbols D, A, O and W, 'frz' and 'svd' (some with subscripts) represent auxiliary state variables that are introduced by the DL2S construction. The user must understand the semantics of these variables as well as the safety property to be proved. Compare this in complexity to the proof by ranking that required the user only to introduce a function that counts the pending time stamps, and needed no auxiliary invariants apart from the simple one needed for safety. Unlike the ranking proof, the DL2S proof can be completed rapidly and automatically by Z3. Unfortunately, we have relieved the burden of undecidability on the automated prover at the cost of a substantial burden of complexity on the user. Our goal here is to eliminate this invidious trade-off.

$$
\begin{aligned}
\forall x.\ \mathrm{pend}(x) &\rightarrow x \leq \mathrm{last} \\
\forall \tau.\mathrm{pend}(\tau) &\rightarrow D(\tau) \\
\forall \tau.\mathrm{frz} \wedge (\mathrm{pend}(\tau) \vee O_{\mathrm{pend}}(\tau)) &\rightarrow A(\tau) \\
\mathrm{svd} \wedge \neg W_{\mathrm{poll}} &\rightarrow \exists \tau.\ (\tau \leq t \wedge O_{\mathrm{pend}}(\tau) \wedge \neg \mathrm{pend}(\tau)) \\
(\mathrm{frz} \vee \neg W_{\neg \mathrm{rsp}}) &\rightarrow \mathrm{pend}(t) \wedge \Box \neg \mathrm{recv}(t) \\
(\neg \mathrm{frz} \wedge W_{\neg \mathrm{rsp}}) &\rightarrow \neg \Box \mathrm{rsp} \\
&\quad\ \Box \Diamond \mathrm{poll} \\
\forall \tau.\ \mathrm{svd} \wedge \tau \leq t \wedge \mathrm{pend}(\tau) &\rightarrow O_{\mathrm{pend}}(\tau) \\
&\quad\ \text{where } \mathrm{rsp} \triangleq \mathrm{send}(t) \rightarrow \Diamond \mathrm{recv}(t)
\end{aligned}
$$

Fig. 2. Invariants for DL2S proof of simple queue example.

3 Relational Rankings

A key point that we failed to notice in the proof of our simple example by well-founded ranking is that the liveness of the time stamped queue does not actually depend on the well-foundedness of the time stamp order. In fact, we could use real numbers for the time stamps, and the queue would still be live. The need for induction over the natural numbers was an artifact of our proof rule, which requires a well-founded ranking. This also explains why the proof by DL2S can be accomplished in EPR – the time stamps are treated as a simple total order, which can be axiomatized within EPR.

This observation suggests a middle path that maintains the conceptual simplicity of the proof by ranking, but eliminates reasoning about well-founded orders other than time. To achieve this, we propose to express rankings as relations over infinite sets, ordered by implication. While this order is not well-founded, we establish finiteness of the ranking relations outside of the logic. This gives us the best of both worlds: we remove the burden of undecidability on the automated prover without the extra burden of complexity imposed on the user by DL2S.

To guarantee that relational rankings are sound for infinite-state systems, we adapt a key idea from the DL2S method to prove that a relation is always finite (that is, its extension as a set of tuples is always finite). To do this, we guarantee that all tuples in the relational ranking are constructed from values that have been produced in a finite computation. This ensures that the ranking relations are finite and therefore that they cannot be infinitely reduced in the implication order.

Our proof approach uses a generic proof rule that is parameterized by a few user-provided relations over the user's program state. The rule avoids exposing the user to the state of a machine-constructed program and still keeps the verification conditions within EPR. For the sake of simple exposition, we will consider a succession of increasingly general rules, culminating in a rule that supports many justice conditions and lexicographic rankings.

3.1 The Relational Reactivity Rule

We begin with a simple proof rule that simulates the Manna and Pnueli reactivity Rule (2). We first consider the problem of proving that some relation R, a function of the program state, is always finite. This can be accomplished by the following proof rule:

$$\frac{\begin{array}{l}\text{F1. } \forall x.\ \neg R(x) \\ \text{F2. } \Box\, \forall x.\ R'(x) \rightarrow (R(x) \vee x = e_1 \vee \cdots \vee x = e_n)\end{array}}{\Box\ R \text{ is finite}} \tag{5}$$

In this rule, $e_1 \ldots e_2$ are expressions that depend on the program state. Typically, they represent the values computed during a single atomic step of the program. This set is necessarily finite. Premise F1 says that the relation R is initially empty. Premise F2 says that at most the elements $e_1 \ldots e_n$ can be added to relation R in each transition. In practice, $e_1 \ldots e_n$ can be just the ground terms occurring in the transition condition. Since a finite number of elements is added to R at each time step, R must be finite at every finite time. Notice we did not state that R is finite in the logic, since this is not expressible in first-order logic.

Now we replace our well-founded ranking δ by a relation, using two shorthands:

$$\begin{aligned}\text{conserves } \delta &\triangleq \forall x.\ \delta'(x) \rightarrow \delta(x) \\ \text{reduces } \delta &\triangleq \exists x.\ \delta(x) \wedge \neg\delta'(x)\end{aligned}$$

That is, a transition 'conserves' the ranking if it does not add any elements, and it 'reduces' the ranking if it removes at least one element. We can now prove formulas of form (1) using the following rule:

$$\frac{\begin{array}{l}\text{C0. } \Box\ R \text{ is finite} \\ \text{C1. } p \Rightarrow (q \vee \phi \wedge (\forall x.\delta(x) \rightarrow R(x))) \\ \text{C2. } \phi \Rightarrow (q' \vee (\phi' \wedge (\text{conserves } \delta)) \\ \text{C3. } \phi \wedge r \Rightarrow (q' \vee (\text{reduces } \delta))\end{array}}{(\Box\Diamond r) \rightarrow (p \Rightarrow \Diamond q)} \tag{6}$$

Premises C1-3 correspond closely to the premises of Rule (2). The key difference is that, in premise C1, we establish that the ranking relation δ is finite at the moment when condition p holds. This is done by establishing that δ is contained in the finite relation R. Using just first-order connectives, we can also express that the ranking is conserved unless q holds (right-hand side of C2) and that the ranking is reduced when r holds (right-hand side of C3). Since δ must be finite while the invariant ϕ holds, it follows that q eventually holds. The advantage of this formulation is that it allows us to express the verification conditions in pure first-order logic, without additional theories. If we take care in the use of quantifier alternations, this allows us to keep the verification conditions within EPR [23].

Now consider proving liveness, of our simple time stamped queue. To prove property (4) using rule (6), we use the following definitions of the predicates in the rule:

$$\begin{aligned}\delta(\tau) &\triangleq \text{pend}(\tau) \wedge \tau \leq t \\ \phi &\triangleq \text{pend}(t)\end{aligned}$$

Notice that the ranking δ is very similar to the one we used above, but it represents simply the set of pending time stamps $\leq t$ and not the cardinality of this set. The invariant ϕ is the same. We can let $R = \delta$, since the program adds only one element to δ at each step, thus finiteness of δ can be proved automatically using Rule (5). This proof has essentially the same conceptual complexity as the proof using well-founded ranking (in fact, slightly less, since we need not define the counting function). However, unlike that proof, it can be checked efficiently and reliably in EPR using Z3, whereas Z3 fails to find a proof for the well-founded ranking. If we make a mistake in the proof, Z3 can produce a true counterexample, guiding us to correct the proof.

3.2 Chaining Liveness Lemmas

Consider proving liveness of a cascade of two queues of the type described above. That is, when we poll queue$_1$, if a time stamp is received, we enter it into queue$_2$. We would like to show an end-to-end response property, that is, assuming $\Box\Diamond \text{poll}_i$ for $i = 1, 2$, we have $\text{send}_1(t) \Rightarrow \Diamond \text{recv}_2(t)$. One way to do this is to prove response properties for the two queues, that is, $\text{send}_i(t) \Rightarrow \Diamond \text{recv}_i(t)$ for $i = 1, 2$ and then use these properties to prove end-to-end response.

Unfortunately, we do not yet have a proof rule that would allow us to prove a response property assuming two response properties. To remedy this, we will relax Rule (6) slightly. First, we first observe that the condition r need not hold true infinitely. It suffices that it hold in the future *until* q is true. Second, we can relax q and q' in the rule's premises to the weaker $\Diamond q$. This gives us the following rule:

$$\frac{\begin{array}{l}\text{C0. } \Box R \text{ is finite} \\ \text{D1. } p \Rightarrow ((\Diamond q) \vee \phi \wedge (\forall x. \delta(x) \rightarrow R(x))) \\ \text{D2. } \phi \Rightarrow ((\Diamond q) \vee (\phi' \wedge (\text{conserves } \delta)) \\ \text{D3. } \phi \wedge r \Rightarrow ((\Diamond q) \vee (\text{reduces } \delta)) \\ \text{D4. } \phi \Rightarrow ((\Diamond q) \vee (\Diamond r))\end{array}}{p \Rightarrow \Diamond q} \tag{7}$$

Notice that we have moved the assumption about r into the premises of the rule, making r effectively a parameter of the rule. Also, observe that the premises now have temporal operators in them (other than the prime symbol, representing 'at the next time'). This may seem to defeat the purpose of a program logic rule, since it does not reduce the proof to ordinary logic. However, we will see that, by appropriate abstractions, we can reduce the temporal verification conditions to decidable propositions.

In our two-queue example, to prove $\text{send}_1(t) \Rightarrow \Diamond \text{recv}_2(t)$, we can use the following values of the rule parameters:

$$
\begin{aligned}
R &\triangleq \text{true} \\
\delta &\triangleq \text{true} \\
\phi &\triangleq \Diamond \text{recv}_1(t) \\
r &\triangleq \text{recv}_1(t) \\
R(\tau) &\triangleq \text{pend}_1(\tau) \vee \text{pend}_2(\tau) \\
\phi &\triangleq \text{pend}_1(t) \vee \text{pend}_2(t)
\end{aligned}
$$

$$
\begin{aligned}
\delta_1(\tau) &\triangleq \text{pend}_1(\tau) \wedge \tau \leq t & \qquad \delta_2(\tau) &\triangleq (\text{pend}_1(\tau) \vee \text{pend}_2(\tau)) \wedge \tau \leq t \\
r_1 &\triangleq \text{poll}_1 & \qquad r_2 &\triangleq \text{poll}_2 \\
\psi_1 &\triangleq \text{pend}_1(t) & \qquad \psi_2 &\triangleq \text{pend}_2(t)
\end{aligned}
$$

Fig. 3. Rule parameters for liveness of the cascaded queues.

We then have to prove the following simplified premises:

$$
\begin{aligned}
&\text{D1. } \text{send}_1(t) \Rightarrow ((\Diamond \text{recv}_2) \vee \phi) \\
&\text{D2. } \phi \Rightarrow ((\Diamond \text{recv}_2) \vee \phi') \\
&\text{D3. } \phi \wedge \text{recv}_1 \Rightarrow (\Diamond \text{recv}_2)) \\
&\text{D4. } \phi \Rightarrow \Diamond \text{recv}_1
\end{aligned}
$$

We can prove D1 using our lemma that $\text{send}_1(t) \Rightarrow \Diamond \text{recv}_1(t)$. This proof is propositional, in the sense that it does not depend on the semantics of the propositions $\Diamond \text{recv}_1$ or $\Diamond \text{recv}_2$. For D2 and D3, we need our lemma $\text{send}_2(t) \Rightarrow \Diamond \text{recv}_2(t)$. That is, if $\text{send}_2(t)$ then $\text{recv}_2(t)$ by the lemma, otherwise $\text{recv}_1(t)$ is false, thus $\Diamond \text{recv}_1(t)$ remains true. The proof for D3 is again purely propositional: the transition relation guarantees that recv_1 implies $\text{send}_2(t)$ which in turn guarantees $\Diamond \text{recv}_2(t)$ by the lemma. We needed one tautology of temporal logic to prove D2, that is, $\Diamond \text{recv}_1 \Rightarrow (\text{recv}_1 \vee (\Diamond \text{recv}_1)')$. This is one of the facts about the $\Diamond$ operator that form the symbolic tableau constraints [3], the other being $\text{recv}_1 \Rightarrow \Diamond \text{recv}_1$. The tableau axioms can be generated automatically from the verification conditions allowing Z3 to discharge the premises. Though this approach is not complete, it allows us to avoid adding further inference rules, for example, to prove single-step eventualities, or to use response properties as assumptions.

Notice that we don't need a ranking for this proof, but we require that receiving on queue_1 and sending on queue_2 occur in the same transition, so that D3 is provable from the tableau axioms. Otherwise we would need a third lemma stating $\text{recv}_1(t) \Rightarrow \Diamond \text{send}_2(t)$. We also need a safety invariant, implying that time stamps in queue_1 are always greater than time stamps in queue_2:

$$\text{last}_2 \leq \text{last}_1 \wedge \forall x.\ \text{pend}_1(x) \rightarrow x \geq \text{last}_2$$

This invariant is also needed to prove the safety property that timestamps are entered into queue_2 in order. Often, invariants needed to prove safety properties are also sufficient for liveness.

3.3 Stable Schedulers

Reducing the proof to many small lemmas, each with its own relational ranking, is an effective approach, but each lemma adds complexity to the proof and opportunities for errors that are time-consuming to correct. Since we require a ranking for each lemma, it would seem more parsimonious, if possible, to combine these rankings into a single ranking and dispense with the statements of the lemmas.

With multiple rankings come multiple justice conditions. In a given state, only one of these must cause the ranking to decrease. We call this justice condition *helpful* in the given state [6]. To establish a justice condition that must eventually reduce the ranking in a given state, we introduce a function called a *stable scheduler*.

As a simple example, consider proving liveness of the cascaded queues above without proving liveness lemmas for the individual queues. It is not difficult to define a ranking over the combined state of the two queues. Let us say that δ_i is the number of time stamps $\tau \leq t$ that are pending in queue$_i$, for $i = 1, 2$. A suitable numeric ranking δ is $2\delta_1 + \delta_2$.

We have two justice conditions to consider, that is, we assume that both queues are polled infinitely often. Unfortunately, neither of these conditions *always* reduces the ranking, since there may be no time stamps $\tau \leq t$ in one or the other of the two queues. One solution to this would be to define a combined justice condition $r \triangleq \text{poll}_1$ if $\delta_1 > 0$ else poll$_2$. This condition would indeed imply that the ranking decreases, but we would have to prove a lemma that it is true infinitely often.

As an alternative, for each justice condition r_i, we provide a *scheduler predicate* ψ_i that determines when r_i is helpful, in the sense of reducing the ranking δ_i, when it eventually holds. We require the scheduler predicates to be *stable*, in the sense that ψ_i, when true, remains true until r_i holds.

A proof rule for response properties with multiple justice conditions and a stable scheduler can be stated as follows:

$$\begin{array}{l}
\text{S1. } p \wedge \neg(\Diamond q) \Rightarrow \phi \\
\text{S2. } \bigwedge_{i=1}^{n} \phi \wedge \neg(\Diamond q) \Rightarrow \\
\qquad \phi' \wedge \\
\qquad (\text{conserves } \delta_i) \wedge \\
\qquad (\psi_i \wedge r_i \rightarrow (\text{reduces } \delta_i)) \wedge \\
\qquad (\psi_i \wedge \neg r_i \rightarrow \psi_i') \\
\text{S3. } \bigwedge_{i=1}^{n} \phi \wedge \neg(\Diamond q) \wedge \psi_i \Rightarrow \Diamond r_i \\
\text{S4. } \phi \wedge \neg(\Diamond q) \Rightarrow \bigvee_{i=1}^{n} \psi_i \\
\text{S5. } \bigwedge_{i=1}^{n} \phi \Rightarrow (\forall x. \delta_i(x) \rightarrow R(x))) \\
\text{S6. } \Box R \text{ is finite} \\
\hline
p \Rightarrow \Diamond q
\end{array} \tag{8}$$

Premise S1 of the rule says that the invariant ϕ is established whenever p is true but the desired eventuality $\Diamond q$ is false. Premise S2 gives several conditions that system transitions must satisfy, in the case invariant ϕ holds, but the eventuality does not. First, the invariant ϕ must be preserved. Second, the each ranking component δ_i must be conserved. Third, if scheduler predicate ψ_i is true and justice condition r_i is true, then the ranking δ_i must be reduced. Fourth, if scheduler predicate ψ_i is true and justice

condition r_i is *false*, then ψ_i must remain true. The last is the stability condition for the scheduler. Premise S3 ensures that every scheduled justice condition eventually occurs. Premise S4 guarantees that at least one justice condition is always scheduled when the invariant holds. Finally, S5 and S6 guarantee that the ranking is always finite while the invariant holds. A proof of soundness of this rule can be found in Appendix A.

For the cascaded queues, we can use the rule parameters shown in Fig. 3. Notice that $\delta_2(\tau)$ is true when time stamp τ appears in either queue, so that moving a time stamp from queue$_1$ to queue$_2$ does not cause δ_2 to increase. Also, notice that δ_2 is reduced when r_2 occurs and ψ_2 holds, because pend$_1(\tau)$ and pend$_2(\tau)$ cannot both be true. This is implied by the safety invariant used above.

Now suppose we change the system so that the queues can hold a bounded number of time stamps. This means that if queue$_2$ is full, we cannot move a time stamp from queue$_1$ to queue$_2$ (in other words, the action that polls queue$_1$ must block). We can prove this system live by a small change to ψ_1 and ψ_2, that is:

$$\psi_1 \triangleq \neg\exists\tau.\ \text{pend}_2(\tau) \wedge \tau \leq t$$
$$\psi_2 \triangleq \exists\tau.\ \text{pend}_2(\tau) \wedge \tau \leq t$$

If there is a time stamp $\tau \leq t$ to send in queue$_2$, then δ_2 must be reduced on polling queue$_2$. Otherwise, by the safety invariant above, queue$_2$ must be empty. Therefore it cannot block, and polling queue$_1$ must reduce δ_1. That is, we design the scheduler to prioritize actions that unblock other actions. To handle a longer cascade of queues, we would prioritize the queues later in the cascade.

3.4 Lexicographic Rankings

Consider now a cascade of queues in which messages can be *reordered*. For example, suppose we have two classes of messages, A and B. We have two corresponding polling actions for queue$_1$: poll$_{1A}$ and poll$_{1B}$. Each action moves the message of its given class with the least time stamp from queue$_1$ to queue$_2$. Thus, messages of different classes can bypass each other in transit. We will assume that queue$_2$ delivers messages in FIFO order, that is, in order of their time of arrival. We will denote the time of arrival of a message with time stamp t at queue$_2$ by a natural number $t_a(t)$.

Reordering of messages presents us with a difficulty in establishing a ranking. That is, at the time a message of class A is sent, we do not have an upper bound on the number of future messages of class B that will bypass it before it reaches queue$_2$. Thus, we cannot establish a finite ranking. One solution to this problem is to take a lemma, such as $\forall\tau.\ \text{send}_1(\tau) \Rightarrow \Diamond\text{pend}_2(\tau)$. At that point when message t reaches queue$_2$, the number of messages in queue$_2$ is known, so we can establish a ranking to prove that message t is eventually received. Alternatively, we could use temporal prophecy to achieve the same effect [20]. However, in both cases we are adding complexity to the proof by asking the user to provide the necessary cut formulas, as well as the rankings. This is unnecessary, however, if we use a lexicographic ranking.

To do this, we introduce a proof rule that combines multiple rankings lexicographically. That is, we introduce a hierarchy of rankings $\delta_1 \ldots \delta_n$, where δ_1 is the high-order

component and δ_n is the low-order component. This induces a lexicographic ranking δ such that $\delta(s_1) < \delta(s_2)$ when $\delta_i(s_1)$ is finite for all $i = 1 \ldots n$, and

$$\vee_{i=1}^{n} \left(\delta_i(s_1) < \delta_i(s_2) \wedge (\wedge_{j=1}^{n-1} \delta_j(s_1) = \delta_j(s_2))\right). \tag{9}$$

We will not, however, explicitly represent δ. Instead, we will establish verification conditions guaranteeing that the δ_i are always finite and that δ is conserved and eventually reduced. For this, a lower-order ranking must be conserved only when no high-order ranking is reduced. Given a scheduler ψ for the rankings, we will say that ranking δ_i is *preempted* if, for some $j < i$, δ_j is scheduled. A preempted ranking need not be reduced, and in fact is allowed to increase, as long as it remains finite. We say that a ranking is *required* if it is scheduled and not preempted. A required ranking must eventually be reduced. This allows us to relax the condition of stability of the schedulers as well, since only the schedulers of required rankings need be stable. We will introduce two shorthands:

$$\begin{aligned} \text{pre}_i(\psi) &= \vee_{j=1}^{i-1} \psi_j \\ \text{req}_i(\psi) &= \psi_i \wedge \neg \text{pre}_i(\psi) \end{aligned}$$

The predicate $\text{pre}_n(\psi)$ indicates that the ranking component δ_i is preempted under scheduler ψ and $\text{req}_n(\psi)$ indicates that δ_i is required.

Using these notations, a suitable rule for establishing liveness with lexicographic relational rankings with stable schedulers is as follows:

$$\begin{array}{l} \text{S1. } p \wedge \neg(\Diamond q) \Rightarrow \phi \\ \text{L2. } \bigwedge_{i=1}^{n} \phi \wedge \neg(\Diamond q) \Rightarrow \\ \qquad \phi' \wedge \\ \qquad (\neg \text{pre}_i(\psi) \rightarrow (\text{conserves } \delta_i)) \wedge \\ \qquad (\text{req}_i(\psi) \wedge r_i \rightarrow (\text{reduces } \delta_i)) \wedge \\ \qquad (\text{req}_i(\psi) \wedge \neg r_i \rightarrow \psi_i') \\ \text{S3. } \bigwedge_{i=1}^{n} \phi \wedge \neg(\Diamond q) \wedge \psi_i \Rightarrow \Diamond r_i \\ \text{S4. } \phi \wedge \neg(\Diamond q) \Rightarrow \bigvee_{i=1}^{n} \psi_i \\ \text{S5. } \bigwedge_{i=1}^{n} \phi \Rightarrow (\forall x. \delta_i(x) \rightarrow R(x))) \\ \text{S6. } \Box R \text{ is finite} \\ \hline p \Rightarrow \Diamond q \end{array} \tag{10}$$

As before, premise S1 establishes the liveness invariant ϕ. Premise L2 differs from S2 in the previous rule in that transitions must conserve a ranking δ_i *only* if it is not preempted, must reduce the ranking when the justice condition occurs *only* if it is required, and must keep the scheduler stable *only* if the ranking is required. The remainder of the premises are the same as in the previous rule. A proof of soundness of this rule can be found in Appendix A.

Now consider again our problem of cascaded queues with reordering, and suppose we want to prove that messages of class A are always eventually received, that is, $\text{send}_1(t) \wedge A(t) \Rightarrow \Diamond \text{recv}_2(t)$, under the assumption the actions poll_{1A} and poll_2 are called infinitely often. This is done with the lexicographic ranking defined in Fig. 4.

Notice that, as long as ψ_1 remains true, executing action poll_{1A} must reduce ranking δ_1, since the earliest class A message must have time stamp $\leq t$. While this is true,

$$R(\tau) \triangleq \text{pend}_1(\tau) \vee \text{pend}_2(\tau)$$
$$\phi \triangleq (\text{pend}_1(t) \vee \text{pend}_2(t)) \wedge A(t)$$

$$\delta_1(\tau) \triangleq \text{pend}_1(\tau) \wedge \tau \leq t \qquad \delta_2(\tau) \triangleq \text{pend}_2(\tau) \wedge t_a(\tau) \leq t_a(t)$$
$$r_1 \triangleq \text{poll}_{1A} \qquad r_2 \triangleq \text{poll}_2$$
$$\psi_1 \triangleq \text{pend}_1(t) \qquad \psi_2 \triangleq \text{pend}_2(t)$$

Fig. 4. Rule parameters for liveness of the cascaded queues with reordering.

the ranking δ_2 is allowed to increase arbitrarily, so long as it remains contained in R. This allows any number of B messages to be added to queue$_2$, bypassing message t. However, when message t is removed from queue$_1$, δ_2 becomes scheduled and must be conserved. This is true because no new messages in queue$_2$ can have arrival times less than $t_a(t)$. The ranking must decrease every time poll$_2$ occurs, since the least arrival time in queue$_2$ must be $\leq t_a(t)$ as long as t remains in the queue. All of the premises of the proof rule are in EPR and can easily be checked automatically.

Parameterized Systems. The are practical situations in which we have an infinite or unbounded number of justice conditions. For example, we may have an infinite or unbounded number of concurrent processes (or to be more precise, process id's may be drawn from an infinite set, if we have dynamic process creation). We wish to assume that every process is scheduled to run infinitely often. In the Apple generic model, this situation arises in various ways, due to fairness assumptions involving unbounded numbers of processors, controllers, addresses and so on. It is straightforward to generalize Rule 10 to parameterized justice conditions. We can do this by simply replacing conjunction and disjunction over finite sets with universal and existential quantification over infinite sets:

$$\begin{array}{l}
\text{S1. } p \wedge \neg(\Diamond q) \Rightarrow \phi \\
\text{P2. } \forall x.\ \bigwedge_{i=1}^{n} \phi \wedge \neg(\Diamond q) \Rightarrow \\
\qquad \phi' \wedge \\
\qquad (\neg \text{pre}_i(\psi)(x) \rightarrow (\text{conserves } \delta_i)) \wedge \\
\qquad (\text{req}_i(\psi_i)(x) \wedge r_i(x) \rightarrow (\text{reduces } \delta_i)) \wedge \\
\qquad (\text{req}_i(\psi)(x) \wedge \neg r_i(x) \rightarrow \psi_i'(x)) \\
\text{P3. } \forall x.\ \bigwedge_{i=1}^{n} \phi \wedge \psi_i(x) \Rightarrow \Diamond r_i(x) \\
\text{P4. } \phi \Rightarrow \exists x.\ \bigvee_{i=1}^{n} \psi_i(x) \\
\text{S5. } \bigwedge_{i=1}^{n} \phi \Rightarrow (\forall x. \delta_i(x) \rightarrow R(x))) \\
\text{S6. } \Box R \text{ is finite} \\
\hline
p \Rightarrow \Diamond q
\end{array} \tag{11}$$

Notice here that premises P2, P3 and P4 are similar to L2, S3 and S4 in Rule (10), except that the predicates ψ_i and r_i have a parameter x that is quantified. Also, notice that n is the number of rankings, not the number of processes, which is conceptually infinite. We can also easily extend the rule to rankings that are n-ary relations rather then unary relations, which is useful in some cases.

Relative Completeness. A temporal proof system is *relatively complete* if it reduces the validity of all temporal propositions to the validity of a finite collection of formulas of arithmetic. We are explicitly *not* concerned with relative completeness here, since our goal is to reduce the proof to *decidable* propositions in EPR *in practice*. Having said this, there are two obvious ways in which the relative incompleteness of our system manifests. First, Rule 5 is not complete for proving finiteness of the ranking in cases where the system can non-deterministically choose an unbounded natural number or set in a single transition, as opposed to computing these values. Second, due to the finiteness requirement, our lexicographic ranking (9) has ordinal height at most ω^n. However, we can describe systems whose reachability relations have greater ordinal height than this. For example, a program that chooses an arbitrary natural number n and descends lexically over the tuples in $\mathbb{N}^n$ has ordinal height ω^ω and hence cannot be proved in our system. This incompleteness has not proved to be an issue, however, our case study.

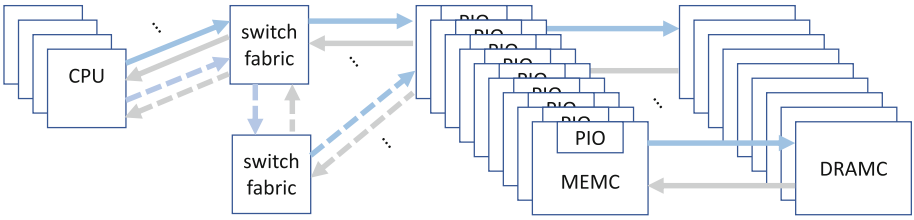

Fig. 5. Apple generic memory model architecture. The flow of memory operations (reads and writes) is depicted by the solid arrows in the diagram, and the flow of I/O operations by dashed arrows. Requests are depicted in light blue, with responses in gray. (Color figure online)

4 Case Study: The Apple Generic Memory Subsystem Model

We now consider applying the relational ranking approach to the Apple generic memory model mentioned in the introduction. Apple provided a generic abstract memory system model that was designed to capture the essential difficulties in proving memory system consistency and liveness without revealing intellectual property. This model has been contributed to the open-source the Ivy project, as have our proofs [1].

The high-level structure of the Apple generic model is sketched in Fig. 5. The system connects a collection of processor cores with a collection of memory and I/O modules, via switch fabrics and controllers. The number of processor cores and controllers is unbounded in the model. At each stage, memory operations (reads and writes) are queued and may be reordered. The order of transmission depends on a set of ordering rules specific to the given stage. In some cases, operations may be blocked by operations that are present in later stages. Whether one operation can bypass another may depend on various attributes of the operations, including the initiating processor, the address, the operation type, and the destination memory controller. While operations may be completed by the memory units out-of-order, they are ultimately retired in-order at the processor cores, by means of a reorder buffer. The Apple engineers developed a

safety proof, using hand-written inductive invariants, that the ordering rules provide a consistent view of memory to the processor cores, according to the desired memory consistency model. The generic model consists of approximately 1200 SLOC in the Ivy language. The safety proof consists of 78 invariants, comprising approximately 500 SLOC.

It was also considered important to prove a liveness property of this model, that is, that every operation issued by a core is eventually retired. This property depends upon a large number of fairness assumptions that constrain the behavior of different units. There are two compelling reasons to verify this specification. First, from a system point of view, we need to know that the ordering rules do not cause any operation to be blocked indefinitely. Second, from a modeling point of view, we need to be certain that the guarantee of memory consistency is not rendered vacuous by a modeling error that prevents certain operations from being retired. However, the DL2S method was found to be too difficult and counterintuitive to be applied in practice for this.

4.1 Liveness Proof with Lemmas

We initially implemented a liveness proof rule in the Ivy tool similar to Rule (8) that allows parameterized justice conditions, without lexicographic ranking. The earliest version of this rule lacked the ability for the user to specify the stable scheduler. Instead, it used a default priority-based scheduler (*i.e.*, the first non-empty ranking is scheduled). Using this implementation, we constructed a liveness proof for the Apple model. The lack of lexicographic ranking made it necessary to break the proof into many lemmas. The proof consists of 26 lemmas expressed in first-order temporal logic, chained together in the style of Sect. 3.2. The basic lemmas are liveness properties relating to the liveness of specific channels in specific modules. Most of these state that an operation reaching one module is eventually transferred to the next module on the appropriate flow path. Additional non-blocking properties are used to state that space eventually becomes available for an operation of a given type in a given module. These basic liveness properties were chained to prove end-to-end liveness.

Most of the lemmas in the proof are universally quantified, typically over module identifiers (*i.e.*, processors or memory controllers) or operation time stamps. When proving the properties, these quantifiers were Herbrandized, that is, replaced by fresh constants. This replacement of variables with constants played a significant role in the construction of rankings, just as the use of a time stamp constant t allowed us to construct a ranking in our simple examples. Moreover, replacing quantified variables with constants also helps in keeping verification conditions within EPR, and thus allowing Ivy to automatically discharge them and produce counterexamples in a reliable way using the Z3 theorem prover.

As an example, the DRAM controllers block incoming operations that have the same address as operations that are already queued. To prove that operations are not indefinitely blocked, we first prove that operation t eventually leaves the DRAM controller, using a ranking. Universally generalizing t, we can then show that all operations in the memory controller eventually reach the DRAM controller, since the blocking operation must eventually depart. Similarly, we first show that the completion of operation timestamped t eventually reaches the reorder buffer at the CPU. Generalizing

t universally, we can then show that every operation u eventually leaves the reorder buffer, since every predecessor of u, for which it waits, must eventually arrive.

We found that it was straightforward to prove all the lemmas save two using the basic relational ranking approach. In all of these cases, Z3 was able to discharge the relevant proof obligations quickly and reliably, without timeouts or divergences. This was critical as each lemma required a few counterexamples to help correct errors in the rule application. Without these counterexamples, developing the proof would have been extremely challenging.

The two lemmas we were unable to prove were instructive. One case involved a queue containing two kinds of operations. We needed to show that if both kinds of operations are removed infinitely often, then every element is eventually removed. The other case was proving that, if completions of all operations eventually reach the reorder buffer (out of order) then all operations are eventually retired (in order). We discharged the two lemmas by model checking a small abstract model (using the eager abstraction method [17]) and then used refinement maps to transfer liveness properties of the small models to the larger model. This method, while effective, was conceptually complex and time-consuming. This experience motivated us to consider the more general proof approach using stable schedulers.

The overall liveness proof consumed approximately 90 person-hours of effort and resulted in fixing several issues in the Apple generic model. The overall textual size of the proof was approximately 1000 SLOC, and all of the proof obligations were checked by Ivy in approximately 115 min on a laptop computer.

4.2 Lemma-Free Proof of Liveness

After adding stable schedulers to the Ivy proof rule, we found that it was straightforward to prove the two lemmas that were previous proved using model checking and refinement relations. However, the complexity of the proof remained high due to the large number of lemmas. Eliminating the lemmas from the proof proved difficult for two reasons. First, as in our simple reordering queue example, the presence of reordering of operations at various places in the system prevented us from expressing the overall ranking as a sum of simple relational rankings. Second, without taking lemmas and Herbrandizing them, we faced the problem of quantifier alternations in the verification conditions that took them outside EPR.

To handle the first problem, we extended the liveness proof rule with lexicographic rankings. This made it possible to express a ranking for the entire end-to-end liveness property without taking lemmas, instead using only a single application of the liveness proof rule. This proof relied on 14 justice assumptions, and for each justice assumption introduced one component in the lexicographic relation ranking.

Checking this proof with Z3 was not possible, however. We found that Z3 produced unpredictable timeouts and was unable to produce the counterexamples needed to debug the proof. The root cause of the problem was quantifier alternations occurring in certain invariants of the system that were needed to prove liveness, but not to prove safety. As a simple example of this phenomenon, consider a mutual exclusion protocol based on ticket numbers, as in [19]. To prove liveness, we must show that every unserved tick number is held by some process that is waiting to enter its critical section.

Otherwise the protocol deadlocks. Deadlock does not affect safety but, of course, does rule out liveness. The invariant we need says that for all ticket numbers t, if t is waiting be served, then some process p holds ticket t. This quantifier alternation introduces a Skolem function from tickets to processes, which forms a function cycle with the map in the system state from processes to ticket numbers. This breaks stratification of the verification conditions, which are thus not in EPR.

Similar situations occur in the Apple generic model. For example, we must show that in every memory module, some time stamp occurs in the first queue position, assuming the queue is not empty. In the proof with lemmas, the quantifier alternations were avoided by simply Herbrandizing the quantifier over memory modules, reducing it to a constant. In the lemma-free proof, however, this was not possible. We found it impractical to carry out the proof with Z3 using non-stratified invariants, because of frequent and unpredictable timeouts.

As an alternative, auxiliary variables were added to the system to provide sufficient witnesses for the offending existential quantifiers, as needed to prove liveness. For example, one auxiliary variable represents the least time stamp present in *any* memory module, and another the identifier of the memory controller module holding this time stamp. The defining properties of these auxiliary variables must be stated as invariants. In some cases, this also entailed more complex ranking definitions.

The textual size of the proof without lemmas is greatly reduced from the proof with lemmas, at about 280 SLOC, of which about 120 represent the auxiliary variables and their invariants, and the checking time is reduced to 15 min (11 for liveness only). The human effort required for the proof was also substantially less, at about 20 h. This figure should be taken with a grain of salt, however, since the second proof effort benefited from the understanding of the system gained in the first.

It is interesting to note that, in the liveness proofs, only one new safety invariant was needed, consisting of a disequality between two variables. The remaining invariants came from the safety proof, or were invariants over the auxiliary variables. The ability to re-use the safety proof greatly reduced the overall effort in proving liveness.

5 Conclusions and Future Work

We have endeavored in this work to develop a method of proving liveness that is is conceptually simple to apply in practice to realistic problems, can be scaled to large problems without modular decomposition, and does not fail unpredictably due to the use of fragile heuristics. No existing method meets these conditions. In a realistic case study, we have seen that relational rankings do. The case study is an of an order of magnitude greater complexity than problems that have been solved by comparable existing methods.

We have also observed that there is a trade-off between the use of lemmas in the proof and the use of lexicographic rankings. The latter approach yielded a proof of lesser textual complexity, but required more sophisticated reasoning to construct the proof, in order to keep the verification conditions within decidable bounds. Handling quantifier alternations in lexicographic proofs is an issue that requires further exploration.

There are several possible directions for further work. One is the problem of assigning root causes to liveness proof failures. One approach would be use state space exploration on the concrete model. For example, we could use model checking to test whether in fact a scheduled justice condition always reduces one of the rankings. If so, the fault likely likes in the safety invariant. Or if the scheduled justice condition implies that the ranking is eventually but not immediately reduced, then an additional ranking may be needed. It is not clear, however, how to effectively explore the state space of complex, infinite-state models to obtain this information.

A related question is automated synthesis of relational rankings. A natural approach is to use a syntax-guided or template-based definition of the search space and to perform the search in a counterexample-guided manner (that is, using CEGIS, or counterexample-guided inductive synthesis). To use CEGIS effectively, we require an effective counterexample diagnosis approach that allows us to rule out large spaces of incorrect proofs. To be useful in practice, such a technique would have to fail transparently, in a way allows effective user guidance.

Another interesting question is whether there are useful classes of distributed systems for which the method is complete, that is, for which there always exist relational rankings that can be verified within EPR.

Finally, while a single realistic case study is useful for motivating and guiding research, it does not allow us to draw conclusions about the general utility of any given method. For this we need a large, representative class of benchmark problems to use in evaluation. Such a benchmark does not currently exist. Developing it would be a significant step toward liveness proof methods that are effective at scale.

A Soundness proofs

In this section, we prove that Rules (8) and (10) are sound. We start by defining the necessary background notions. We use standard multi-sorted first order logic. If s is a multi-sorted first-order structure, we write $\sigma[s]$ for the universe of sort σ in structure s, and $\phi[s]$ for the interpretation of formula ϕ in structure s. For the sake of notational simplicity, we restrict our attention in the sequel to unary relations, but the extension to n-ary relations is straightforward. A (unary) relation over sort σ is a formula ψ of the form $\lambda x.\ p(x)$, where p is a first-order formula whose only free variable is x and x is of sort σ. We write $\psi[s]$ for the function that takes x to $p(x)[s]$. We adopt the standard semantics of first-order linear temporal logic with the prime operator, so that t' indicates the value of t at the next time. Moreover, we take the axiom $\mathcal{I} \wedge \Box\mathcal{T}$ (where $\langle \mathcal{I}, \mathcal{T} \rangle$ is intended to represent a transition system with initial condition $\mathcal{I}$ and transition condition $\mathcal{T}$).

Definition 1. *A* lexicographic relational ranking *is an indexed set (possibly empty) of unary predicates, $\delta = \{\delta_{i=1\ldots n}\}$, where each predicate may be over a different sort. We say δ is* finite *in structure s if $\delta_i[s]$ is finite for all $i = 1 \ldots n$. The ranking on structures induced by δ is the pre-order $<_\delta$ such that, for any two structures $s_{1,2}$ over the same universe, $s_1 <_\delta s_2$ iff:*

- *δ is finite in s_1, and*

– *for some* $i \in 1 \ldots n$, $\delta_i[s_1] \subset \delta_i[s_2]$ *and for all* $1 \leq j < i$, $\delta_j[s_1] = \delta_j[s_2]$.

Theorem 1. *For any lexicographic relational ranking* $\delta = \{\delta_{i=1\ldots n}\}$, *the pre-order* $<_\delta$ *is well-founded.*

Proof. By induction on n. In the base case, $n = 0$, the order is well-founded because it is empty. In the induction step, we show that if there is an infinite downward chain in the order $<_\delta$, there is an infinite downward chain in the order $<_\epsilon$ where ϵ is the ranking $\{\epsilon_{i=1\ldots n-1}\}$ such that $\epsilon_i = \delta_{i+1}$. This is a contradiction, since by inductive hypothesis, ϵ is well-founded. To see this, suppose we have an infinite descending chain $t_0 >_\delta t_1 >_\delta \cdots$. By definition, δ must be finite in t_1. Moreover, for all $i = 1 \ldots$, we must have $\delta_1[t_i] \supseteq \delta_1[t_{i+1}]$. Since $\delta_1[t_1]$ is finite, in cannot infinitely decrease, therefore there exists an i such that $\delta_1[t_j] = \delta_1[t_{j+1}]$ for all $j \geq i$. It follows that the sequence $t_j, t_{j+1}, \ldots$ is an infinite descending chain of ϵ. □

Theorem 2. *Rule (10) is sound.*

Proof. Suppose toward a contradiction that there exists a sequence of structures $s = s_o, s_1, \ldots$ satisfying $\mathcal{I} \wedge \Box\mathcal{T}$, such that $p[s_i]$ is true for some i, but $q[s_j]$ is false for all $j \geq i$, and all the premises of the rule hold in s. From premise S1 we have $\phi[s_i]$ and from L2, by induction on time, that $\phi[s_j]$ for all $j \geq i$. By S5 and S6, we know that delta is finite in s_j for all $j \geq i$. Now we show that for all $j \geq i$, there exists a $k > j$ such that $s_j >_\delta s_k$. By S4, we know that there exists an $l \in 1 \ldots n$ such that $\psi_i[s_j]$. Therefore, let l be the *least* number in $1 \ldots n$ such that, for *some* $m \geq j$, we have $\psi_l[s_m]$. We have $\neg \mathrm{pre}_l(\psi)[s_k]$ for all $k \geq j$ and therefore by L2 and induction, $\delta_l(s_j) \supseteq \delta_l(s_k)$. From S3, it follows that there exists $k \geq j$ such that $r_i[s_k]$. Moreover, by L2 and induction on time, we have, for all $j \leq m \leq k$, $\mathrm{req}_l(\psi)[s_m]$ and $\psi_l[s_m]$. Thus, by L2 we have $\delta_l[s_k] \supset \delta_l[s_{k+1}]$, hence $\delta_l[s_j] \supset \delta_l[s_{k+1}]$. We have proved that for all $j \geq i$, there exists a $k > j$ such that $s_j >_\delta s_k$. It follows that there exists an infinite descending chain in $<_\delta$, which contradicts Theorem 1. □

Theorem 3. *Rule (8) is sound.*

Proof. Since premise S2 of Rule (8) implies premise L2 of Rule (10), and the remaining premises of the two rules are identical, it follows by Theorem 3 that Rule (8) is sound.

References

1. Apple, Inc. Apple Generic Memory Model. https://github.com/kenmcmil/ivy/tree/master/doc/examples/apple
2. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. **66**(2), 160–177 (2002)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS, pp. 428–439. IEEE Computer Society (1990)
4. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_8
5. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS, pp. 337–340 (2008)

6. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with invisible ranking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 223–238. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_19
7. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with incomprehensible ranking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 482–496. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_36
8. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
9. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in Satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
10. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
11. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand (2021). CoRR, abs/2106.00664
12. Hawblitzel, C., et al.: IronFleet: proving safety and liveness of practical distributed systems. Commun. ACM, **60**(7), 83–92 (2017)
13. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs (2014)
14. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
15. Manna, Z., Pnueli, A.: Completing the temporal picture. Theor. Comput. Sci. **83**(1), 91–130 (1991)
16. McMillan, K.L.: Circular compositional reasoning about liveness. In: Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings, pp. 342–345 (1999)
17. McMillan, K.L.: Eager abstraction for symbolic model checking. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 191–208. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_11
18. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 190–202. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_12
19. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. Proc. ACM Program. Lang. **2**(POPL), 26:1–26:33 (2018)
20. Padon, O., Hoenicke, J., McMillan, K.L., Podelski, A., Sagiv, M., Shoham, S.: Temporal prophecy for proving temporal properties of infinite-state systems. In: FMCAD, pp. 1–11. IEEE (2018)
21. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA), 108:1–108:31 (2017)
22. Ramsey, F.: On a problem in formal logic. Proc. London Math. Soc. (1930)
23. Marcelo Taube, et al.: Modularity for decidability of deductive verification with applications to distributed systems. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, pp. 662–677. ACM (2018)
24. Yao, J., Tao, R., Gu, R., Nieh, J.: Mostly automated verification of liveness properties for distributed protocols with ranking functions. In: POPL (2024). To appear