On the Refinement of Liveness Properties of Distributed Systems[1]

Paul Attie

Department of Computer Science and Center for Advanced Mathematical Studies
American University of Beirut
P.O. Box 11-0236
Riad El Solh
Beirut 1107 2020
Lebanon
pa07@aub.edu.lb
Tel: +961 1 350 000 ext. 4221
Fax: +961 1 744 461

May 3, 2010

Keywords: automata, simulation relations, liveness, refinement, proof diagrams.

Abstract We present a new approach, based on simulation relations, for reasoning about liveness properties of distributed systems. Our contribution consists of (1) a formalism for defining liveness properties, (2) a proof method for liveness properties based on that formalism, and (3) two expressive completeness results: our formalism can express any liveness property which satisfies a natural "robustness" condition; and also any liveness property at all, provided that history variables can be used.

To define liveness, we generalize complemented-pairs (Streett) automata to an infinite state-space, and an infinite number of complemented-pairs. Our proof method provides two techniques: one for refining liveness properties across levels of abstraction, and another for refining liveness properties within a level of abstraction. The first is based on extending simulation relations so that they relate the liveness properties of an abstract automaton to those of a concrete automaton. The second is based on a deductive method for inferring new liveness properties of an automaton from already established liveness properties of the same automaton. This deductive method is diagrammatic, and is based on constructing "lattices" of liveness properties.

1 Introduction and Overview

One of the major approaches to the construction of correct distributed systems is the use of an operational specification, e.g., an *automaton* or a *labeled transition system*, which is successively refined, via several intermediate levels of abstraction, into an implementation. The implementation is considered correct if and only if each of its externally visible behaviors, i.e., *traces*, is also a trace of the specification. This "trace inclusion" of the implementation in the specification is usually established transitively by means of establishing the trace inclusion of the system description at each level of abstraction in the system description at the next higher level. When reasoning at

[1] Some of the results in this paper appeared in the eighteenth ACM Symposium on Principles of Distributed Computing, (PODC'99), under the title "Liveness-preserving Simulation Relations".

any particular level, we call the lower level the concrete level, and the higher level the abstract level.

The correctness properties of a distributed system are classified into *safety* and *liveness* [28]: safety properties state that “nothing bad happens,” for example, that a database system never produces incorrect responses to queries, while liveness properties state that “progress occurs in the system,” for example, every query sent to a database system is eventually responded to. Safety properties are characterized by the fact that they are violated in finite time: e.g., once a database has returned an incorrect response to an external user, there is no way to recover to where the safety property is satisfied. Liveness properties, on the other hand, are characterized by the fact that there is always the possibility of satisfying them: the database always has the opportunity of responding to pending queries. Thus, an operational specification defines the required safety properties by means of an automaton, or labeled transition system. The reachable states and transitions of the automaton are the “good” states/transitions, whose occurrence does not violate safety. Any unreachable states, if present, are “bad,” i.e., they represent a violation of the safety properties, e.g., due to a fault. The occurrence of such a “bad” state is something that happens in finite time, and so constitutes the violation of a safety property. The liveness properties are specified by designating a subset of the executions of the automaton as being the “live” executions, leading to the notion of *live execution property.* These are the executions along which eventually, all the necessary actions are executed, e.g., the actions that respond to pending queries. To express the idea that there is always the possibility of satisfying a liveness property, this subset of the executions must have the property that any finite execution can be extended to an execution in the subset [1].

Distributed systems consist of many sequential processes which execute concurrently. To reason effectively about such large systems, researchers have proposed the use of *compositional reasoning*: global properties of the entire system are inferred by first deducing local properties of the constituent processes or subsystems, and then combining these local properties to establish the global properties. In particular, we desire that refinement is compositional: when a particular process P_i is refined to a new process P_i', we wish to reason only about whether P_i' is a correct refinement of P_i, without having to engage in global reasoning involving all of the other processes in the system. The need for compositional reasoning, as well as notions such as behavioral subtyping [31] and information hiding, motivated the development of the notion of *externally visible behavior*, e.g., the set of traces of an automaton, where a trace is a sequence of “external” actions, visible at the interface, which the automaton can engage in. Typically, a trace is obtained by taking an execution and removing all the *internal* information, i.e., the states and the internal actions.

The notion of externally visible behavior then leads naturally to notions of external safety and liveness properties, which are specified over the traces of an automaton, rather than over the (internal) states and executions. The external safety property is the set of all traces, since this is the external “projection” of all the executions, which define the reachable states and transitions, which in turn give us the safety properties, as discussed above. The external liveness property is obtained by taking the traces of all the live executions. These are called the *live traces*, and the set of all live traces is a *live trace property.*

Trace inclusion usually means that every trace of the concrete automaton is a trace of the abstract automaton. Thus, trace inclusion deals with safety properties: every safety property of the set of traces of the concrete automaton is also a safety property of the set of traces of the abstract automaton. Thus, external safety properties are preserved by the refinement from the abstract to the concrete. Trace inclusion does not address liveness properties, however. The appropriate notion of inclusion for external liveness properties is *live trace inclusion* [18, 19]: every live trace of the concrete automaton is a live trace of the abstract automaton.

Consider again the database example, with the external liveness property that every query submitted is eventually processed. Let B be a high-level specification of such a system. By using state variables that record requests and responses, this property can be easily stated in terms of the executions of B, which results in a live execution property. The set of traces of the live executions then gives the corresponding live trace property. Provided that the state variables which record requests and responses are updated correctly, the live trace property will only contain traces in which every input of a query to the database (e.g., from an external "user") is eventually followed by an output of a response from the database (to the user).

Let A be an implementation of B. The live executions of A are defined by the liveness properties that typically can be guaranteed by reasonable implementations, e.g., "fair scheduling" [16]—every continuously enabled action (or process) is eventually executed, and fair polling of message channels—every message sent is eventually received[2]. The set of traces of the live executions then gives the live trace property corresponding to this action/process fairness and reliable message delivery in the underlying execution behavior. However, the live trace property that we wish to verify for A is not this property per se, but the same live trace property which B has, namely that every input of a query to the database is eventually followed by an appropriate output from the database. This paper addresses the problem of verifying such liveness properties for an implementation A.

It is clear that verifying that the live traces of A are contained in the live traces of B immediately yields the desired conclusion, namely that A has the desired live trace property. Thus, live trace inclusion applied to the above example implies that every trace of an execution of A in which all messages sent are eventually received, and all continuously enabled actions (processes) are eventually executed, i.e., a live trace of A, is also a live trace of B, i.e., a trace in which all queries receive a response. This is exactly what is required, since the liveness properties of A along executions where, for example, messages sent are not received, are not of interest. Conversely, a live execution of A, in which all messages sent are received, and scheduling is fair, should produce an external behavior which has the desired liveness properties: every query receives a response. More generally, live trace inclusion implies that external liveness properties are preserved by the refinement from the specification B to the implementation A.

One of the main proof techniques for establishing trace inclusion is that of establishing a *simulation* [35] or *bisimulation* [44] between the concrete and the abstract automata. A simulation (or bisimulation) establishes a certain correspondence (depending on the precise type of simulation) between the states/transitions of the concrete automaton and the states/transitions of the abstract automaton, which then implies trace inclusion. An important advantage of the simulation-based approach is that it only requires reasoning about individual states and finite execution fragments, rather than reasoning about entire (infinite) executions. Unfortunately, the end-result, namely

[2] We do not address fault-tolerance for the time being, thus messages are always received along a live execution. See Section 7.2 for a discussion of how the techniques presented in this paper can be applied to fault-tolerance.

the establishment of trace inclusion, does not, as we establish in the sequel, imply live trace inclusion, since the set of live traces is, in general, a proper subset of the set of traces.

Our contributions. In this paper, we show how to use simulation relations to reason about liveness. Our approach uses a state-based technique to specify live execution properties: a *liveness condition* is given as a (possibly infinite) set of ordered pairs $\langle \mathsf{Red}_i, \mathsf{Green}_i \rangle$, where Red_i, Green_i are sets of states. An execution is considered to satisfy a single pair $\langle \mathsf{Red}, \mathsf{Green} \rangle$ iff whenever it contains infinitely many states in Red, then it also contains infinitely many states in Green. An execution is live iff it satisfies all the pairs in the liveness condition. A trace is live iff it is the trace of some live execution. Our notion of liveness condition is akin to the acceptance condition of a *complemented-pairs (or Streett) automaton* [4,14,21], except that we allow an infinite number of pairs, and also our automata can have an infinite number of states and transitions. We then present the notion of *liveness-preserving simulation relation*, which appropriately relates the states mentioned in the concrete automaton's liveness condition to those mentioned in the abstract automaton's liveness condition. This is done in two stages. The first stage refines the liveness condition of the abstract automaton into a "derived" liveness condition of the concrete automaton. This derived condition may contain complemented-pairs that are not directly specified in the liveness condition of the concrete automaton. The second stage then proves that the derived condition is implied by the directly specified liveness condition of the concrete automaton (using a "lattice" construction). The use of such a derived liveness condition allows us to break down the refinement problem at each level into two simpler subproblems, since the derived liveness condition of the concrete automaton can usually be formulated to better match with the liveness condition of the abstract automaton. Establishing a liveness-preserving simulation relation then allows us to conclude that every live trace of the concrete automaton is also a live trace of the abstract automaton. As discussed above, our method can be applied to multiple levels of abstraction, where the specification is successively refined in stages, producing several intermediate descriptions of the specified system, until a description that is directly implementable on the desired target architecture and has adequate performance and fault-tolerance properties is derived. Thus, we address the problem of preserving liveness properties in the successive refinement of a specification into an implementation, which contributes to making the method scalable, as our extended example in Section 6 shows.

We establish two expressive completeness results for complemented-pairs liveness conditions. The first shows that any live execution property which satisfies a natural "robustness" condition can be specified by a complemented-pairs liveness condition. The second shows that any live execution property whatsoever can be specified by a complemented-pairs liveness condition, provided that history variables can be used.

The paper is organized as follows. Section 2 provides technical background on automata and simulation relations from [18] and [35]. Section 3 gives our key technical notion of a live automaton, i.e., an automaton equipped with a liveness condition, and also defines live executions, live traces, and derived liveness properties. Section 4 presents our definitions for liveness-preserving simulation relations, and shows that liveness-preserving simulation relations imply live trace inclusion. Section 5 shows how a derived liveness condition can be deduced from the directly specified condition. Together, these two sections give our method for refining liveness properties. Section 6 applies our results to the eventually-serializable data service of [15,27]. Section 7 examines some alternative choices for expressing liveness, shows that our method can also be applied to fault-tolerance properties, and briefly discusses the mechanization of our method. Section 8 discusses the expressiveness of complemented-pairs for liveness properties, and presents two relative completeness results. Section 9 discusses related work. Finally, Section 10 presents our conclusions and discusses avenues for further research. Appendix A gives some background on simulation relations, Appendix B gives some background on temporal logic, and Appendix C presents I/O automaton pseudocode for the eventually-serializable data service of [15,27].

2 Technical Background

The definitions and theorems in this section are taken from [18] and [35], to which the reader is referred for details and proofs.

2.1 Automata

Definition 1 (Automaton) An *automaton* A consists of four components:

1. a set $\mathit{states}(A)$ of states,
2. a nonempty set $\mathit{start}(A) \subseteq \mathit{states}(A)$ of start states,
3. an action signature $\mathit{sig}(A) = (\mathit{ext}(A), \mathit{int}(A))$ where $\mathit{ext}(A)$ and $\mathit{int}(A)$ are disjoint sets of external and internal actions, respectively (let $\mathit{acts}(A)$ denote the set $\mathit{ext}(A) \cup \mathit{int}(A)$), and
4. a transition relation $\mathit{steps}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A)$.

Let $s, s', u, u', \ldots$ range over states and $a, b, \ldots$ range over actions. Write $s \xrightarrow{a}_A s'$ iff $(s, a, s') \in \mathit{steps}(A)$. We say that a is *enabled* in s. An execution fragment α of automaton A is an alternating sequence of states and actions $s_0 a_1 s_1 a_2 s_2 \ldots$ such that $(s_i, a_{i+1}, s_{i+1}) \in \mathit{steps}(A)$ for all $i \geq 0$, i.e., α conforms to the transition relation of A. Furthermore, if α is finite then it ends in a state. If α is an execution fragment, then $\mathit{fstate}(\alpha)$ is the first state along α, and if α is finite, then $\mathit{lstate}(\alpha)$ is the last state along α. If α_1 is a finite execution fragment, α_2 is an execution fragment, and $\mathit{lstate}(\alpha_1) = \mathit{fstate}(\alpha_2)$, then $\alpha_1 {}^\frown \alpha_2$ is the concatenation of α_1 and α_2 (with $\mathit{lstate}(\alpha_1)$ repeated only once). Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an execution fragment. Then the length of α, denoted $|\alpha|$, is the number of actions in α. $|\alpha|$ is infinite if α is infinite, and $|\alpha| = 0$ if α consists of a single state. Also, $\alpha|_i \overset{\mathrm{df}}{=} s_0 a_1 s_1 \ldots a_i s_i$. If α is a prefix of α', we write $\alpha \leq \alpha'$. We also write $\alpha < \alpha'$ for $\alpha \leq \alpha'$ and $\alpha \neq \alpha'$.

An execution of A is an execution fragment that begins with a state in $\mathit{start}(A)$. The set of all executions of A is denoted by $\mathit{execs}(A)$, and the set of all infinite executions of A is denoted by $\mathit{execs}^\omega(A)$. A state of A is *reachable* iff it occurs in some execution of A. The trace $\mathit{trace}(\alpha)$ of execution fragment α is obtained by removing all the states and internal actions from α. The set of traces of an automaton A is defined as the set of traces β such that β is the trace of some execution of A. It is denoted by $\mathit{traces}(A)$. If φ is a set of executions, then $\mathit{traces}(\varphi)$ is the set of traces β such that β is the trace of some execution in φ. If a is an action, then we define $\mathit{trace}(a) = a$ if a is external, and $\mathit{trace}(a) = \lambda$ (the empty sequence) if a is internal. If $a_1 a_2 \cdots a_n$ is a sequence of actions, then $\mathit{trace}(a_1 \cdots a_n) = \mathit{trace}(a_1)\mathit{trace}(a_2) \cdots \mathit{trace}(a_n)$, where juxtaposition denotes concatenation.

If R is a relation over $S_1 \times S_2$ (i.e., $R \subseteq S_1 \times S_2$) and $s_1 \in S_1$, then we define $R[s_1] = \{s_2 \mid (s_1, s_2) \in R\}$. We use $\upharpoonright$ to denote the restriction of a mapping to a subset of its domain.

2.2 Simulation Relations

We shall study two different simulation relations: forward simulations [35] and backward simulations [52]. The technical report version of this paper [5] extends the results to three other simulation relations: refinement mappings, history relations, and prophecy relations. These relations all preserve safety properties. In Section 4, we extend these simulation relations so that they preserve liveness as well as safety. A forward simulation requires that (1) each execution of an external action a of A is matched by a finite execution fragment of B containing a, and all of whose other actions are internal to B, and (2) each execution of an internal action of A is matched by a finite (possibly empty) execution fragment of B all of whose actions are internal to B (if the fragment is empty, then we have $u \in f[s']$, i.e., u and s' must be related by the simulation). It follows that forward simulation implies trace inclusion (also referred to as the *safe preorder* below), i.e., if there is a forward simulation from A to B, then $\mathit{traces}(A) \subseteq \mathit{traces}(B)$. Likewise, the other simulation relations all imply trace inclusion (the backward simulation and prophecy relation must be image-finite) for similar reasons. See Lemma 6.16 in [18] for a formal proof of this result.

We use F to denote forward simulation, and iB to denote image-finite backward simulation. We write $X \in \{F, iB\}$ to mean that X is one of these two relations. We write $A \leq_F B$ if there exists a forward simulation from A to B w.r.t. some invariants, and $A \leq_F B$ via f if f is a forward simulation from A to B w.r.t. some invariants. Similarly for image-finite backward simulation. Appendix A gives formal definitions for forward simulation and image-finite backward simulation.

2.3 Execution Correspondence

Simulation relations induce a correspondence between the executions of the concrete and the abstract automata. This correspondence is captured by the notion of R-relation. If $\alpha' = u_0 b_1 u_1 b_2 u_2 \cdots$ is an execution of automaton B, then define $\mathit{trace}(\alpha', j, k)$ to be $\mathit{trace}(b_j \cdots b_k)$ if $j \leq k$, and to be λ (the empty sequence) if $j > k$.

Definition 2 (R-relation and Index Mappings) Let A and B be automata with the same external actions and let R be a relation over $\mathit{states}(A) \times \mathit{states}(B)$. Furthermore, let α and α' be executions of A and B, respectively:

$\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$
$\alpha' = u_0 b_1 u_1 b_2 u_2 \cdots$

Say that α and α' are *R-related*, written $(\alpha, \alpha') \in R$, if there exists a total, nondecreasing mapping $m : \{0, 1, \ldots, |\alpha|\} \mapsto \{0, 1, \ldots, |\alpha'|\}$ such that:

1. $m(0) = 0$,
2. $(s_i, u_{m(i)}) \in R$ for all i, $0 \leq i \leq |\alpha|$,
3. $\mathit{trace}(\alpha', m(i-1)+1, m(i)) = \mathit{trace}(a_i)$ for all i, $0 < i \leq |\alpha|$, and
4. for all $j, 0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$.

The mapping m is referred to as an *index mapping* from α to α' with respect to R. Write $(A, B) \in R$ if for every execution α of A, there exists an execution α' of B such that $(\alpha, \alpha') \in R$.

Theorem 1 (Execution Correspondence Theorem) *Let A and B be automata with the same external actions. Suppose $A \leq_X B$ via S, where $X \in \{F, R, iB, H, iP\}$. Then $(A, B) \in S$.*

Lemma 1 *Let A and B be automata with the same external actions and let R be a relation over $states(A) \times states(B)$. If $(\alpha, \alpha') \in R$, then $trace(\alpha) = trace(\alpha')$.*

Theorem 1 and Lemma 1 appear in [18] as Theorem 6.11 and Lemma 6.15, respectively.

2.4 Linear-time Temporal Logic

We use the fragment of linear-time temporal logic consisting of the $\Box$ (always) and $\Diamond$ (eventually) operators over state assertions [48,40]. In particular, we use the "infinitary" operators $\Box\Diamond$ (infinitely often) and $\Diamond\Box$ (eventually always). We specify state assertions as a set of states, the state in question satisfying the assertion iff it belongs to the set.

For example, if U is a set of states, then $\alpha \models \Box\Diamond U$ means "α contains infinitely many states from U," and $\alpha \models \Diamond\Box U$ means "all but a finite number of states of α are from U." These operators can be combined with propositional connectives $(\neg, \wedge, \vee, \Rightarrow)$ so that, for example, $\alpha \models \Box\Diamond U' \Rightarrow \Box\Diamond U''$ means "if α contains infinitely many states from U', then it also contains infinitely many states from U'', and $\alpha \models \Diamond\Box\neg U$ means "all but a finite number of states of α are not from U."

Appendix B provides a formal definition of the syntax and semantics of the temporal logic that we use.

3 Live Automata

We first formalize the notions of live execution property and live trace property.

Definition 3 (Live Execution Property) Let A be an automaton, and $\varphi \subseteq execs^{\omega}(A)$. Then, φ is a *live execution property* for A if and only if for every finite execution α of A, there exists an infinite execution α' of A such that $\alpha < \alpha'$ and $\alpha' \in \varphi$.

In other words, a live execution property is a set of infinite executions of A such that every finite execution of A can be extended to an infinite execution in the set. This requirement was proposed in [1], where it is called *machine closure*.

Note that we do not consider interaction with an environment in this paper. This is why we use automata rather than I/O automata, i.e., we have external actions without an input/output distinction. This issue is treated in detail in [19], where a liveness property is defined as a set of executions (finite or infinite) such that any finite execution can be extended to an execution in the set. Thus, an extension may be finite, unlike our approach. This is because requiring extension to an infinite execution may constrain the environment: an execution ending in a state with no enabled internal or output action will then require the environment to execute an action that is an output of the environment and an input of the automaton, so that the execution can be extended to an infinite one. We defer treating this issue to another occasion.

Definition 4 (Live Trace Property) Let A be an automaton, and $\psi \subseteq \mathit{traces}(A)$. Then, ψ is a *live trace property* for A if and only if there exists a live execution property φ for A such that $\psi = \mathit{traces}(\varphi)$.

In [18,19], the notion of live execution property was the basic liveness notion, and a live automaton was defined to be an automaton A together with a live execution property. This use of an arbitrary set of executions as a liveness property, subject only to the machine closure constraint resulted in a proof method in [18] which requires reasoning over entire executions. Since we wish to avoid this, we take as our basic liveness notion the *complemented-pairs condition* of Streett automata, with the proviso that we extend it to an infinite state-space and an infinite number of complemented-pairs. In the next section, we show that this approach to specifying liveness entails no loss of expressiveness, provided that we can use history variables.

Let A be an automaton. We say that p is a *complemented-pair*[3] over A iff p is an ordered pair $\langle\!\langle \mathsf{Red}, \mathsf{Green} \rangle\!\rangle$ where $\mathsf{Red} \subseteq \mathit{states}(A)$, $\mathsf{Green} \subseteq \mathit{states}(A)$. Given $p = \langle\!\langle \mathsf{Red}, \mathsf{Green} \rangle\!\rangle$, we define the selectors $p.\mathsf{R} = \mathsf{Red}$ and $p.\mathsf{G} = \mathsf{Green}$. Let α be an infinite execution of A. Then, we write $\alpha \models \langle\!\langle \mathsf{Red}, \mathsf{Green} \rangle\!\rangle$ iff $\alpha \models \Box\Diamond\mathsf{Red} \Rightarrow \Box\Diamond\mathsf{Green}$, i.e., if α contains infinitely many states in Red, then it also contains infinitely many states in Green. We also write $\alpha \models p$ in this case. Our goal is a method for refining liveness properties using reasoning over states and finite execution fragments only, in particular, avoiding reasoning over entire (infinite) executions. We therefore formulate a liveness condition based on states rather than executions.

Definition 5 (Live Automaton with Complemented-pairs Liveness Condition) A *live automaton* is a pair (A, L) where:

1. A is an automaton, and
2. L is a set of pairs $\{\langle\!\langle \mathsf{Red}^i_A, \mathsf{Green}^i_A \rangle\!\rangle \mid i \in \eta\}$ where $\mathsf{Red}^i_A \subseteq \mathit{states}(A)$ and $\mathsf{Green}^i_A \subseteq \mathit{states}(A)$ for all $i \in \eta$, and η is some cardinal, which serves as an index set,

   and A, L satisfy the following constraint:

– for every finite execution α of A, there exists an infinite execution α' of A such that
  $\alpha < \alpha'$ and $(\forall p \in L : \alpha' \models p)$.

(A, L) inherits all of the attributes of A, namely the states, start states, action signature, and transition relation of A. The executions (execution fragments) of (A, L) are the executions (execution fragments) of A, respectively. We say that L is a *complemented-pairs liveness condition* over A. Often we use just "liveness condition" instead of "complemented-pairs liveness condition."

The constraint in Definition 5 is the machine closure requirement, that every finite execution can be extended to a live execution.

Definition 6 (Live Execution) Let (A, L) be a live automaton. An execution α of (A, L) is a *live execution* iff α is infinite and $\forall p \in L : \alpha \models p$.
We define $\mathit{lexecs}(A, L) = \{\alpha \mid \alpha \in \mathit{execs}^\omega(A) \text{ and } (\forall p \in L : \alpha \models p)\}$.

[3] When it is clear from context, we just say "pair".

Our notion of liveness condition is essentially the acceptance condition for finite-state complemented-pairs automata on infinite strings [14], with the important difference that we generalize it to an arbitrary (possibly infinite) state space, and allow a possibly infinite set of pairs. Despite the possibility that Red^i_A and Green^i_A are infinite sets of states, it is nevertheless very convenient to have an infinite number of complemented-pairs. Using the database example of the introduction, we can express the liveness property "every query submitted is eventually processed" as the infinite set of pairs $\{\langle x \in \mathit{wait}, x \notin \mathit{wait} \rangle \mid x \text{ is a query}\}$, and where *wait* is the set of all queries that have been submitted but not yet processed (x is removed from *wait* when it is processed). Being able to allocate one pair for each query facilitates the very straightforward expression of this liveness property. Our extended example in Section 6 also uses an infinite number of pairs in this manner.

The above discussion applies to any system in which there are an infinite number of *distinguished* operations, e.g., each operation has a unique identifier, as opposed to, for example mutual exclusion for a fixed finite number of processes,, where there are an infinite number of entries into the critical section by some process P_i, but these need not be "distinguished," since the single liveness property $\Box(\mathit{request}(P_i) \Rightarrow \Diamond \mathit{critical}(P_i))$ is sufficient to account for all of these. The key point is that only a bounded number of outstanding requests must be dealt with ($\leq$ the number of processes) , whereas in a system in which there are an infinite number of distinguished operations, an unbounded number of outstanding requests must be dealt with. We conjecture that the liveness property "every request is eventually satisfied" cannot even be stated using a finite number of complemented pairs.

The safe preorder, live preorder [18] embody our notions of correct implementation with respect to safety, liveness, respectively.

Definition 7 (Safe preorder, Live preorder) Let (A, L), (B, M) be live automata with the same external actions ($\mathit{ext}(A) = \mathit{ext}(B)$). We define:

Safe preorder: $(A, L) \sqsubseteq_s (B, M)$ iff $\mathit{traces}(A) \subseteq \mathit{traces}(B)$

Live preorder: $(A, L) \sqsubseteq_\ell (B, M)$ iff $\mathit{traces}(\mathit{lexecs}(A, L)) \subseteq \mathit{traces}(\mathit{lexecs}(B, M))$

From [35,18], we have that simulation relations imply the safe preorder, i.e., if $A \leq_X B$ where $X \in \{F, iB\}$, then $(A, L) \sqsubseteq_s (B, M)$.

Returning to the database example of the introduction, if α is some live execution of the implementation A, then, along α, every continuously enabled action is eventually executed (action fairness) and every message sent is eventually received (message fairness). The trace β of α is then an externally visible live behavior of A: $\beta \in \mathit{traces}(\mathit{lexecs}(A, L))$. If A is a correct implementation, then we expect that the enforcement of action fairness and message fairness in A then guarantees the required liveness properties of the specification, namely that every query is eventually processed. Thus, the externally visible live behavior β of A must satisfy the required liveness properties of the specification, i.e., $\beta \in \mathit{traces}(\mathit{lexecs}(B, M))$. This is exactly what the live preorder requires.

Definition 8 (Semantic Closure of a Liveness Condition) Let (A, L) be a live automaton. The semantic closure $\widehat{L}$ of L in A is given by $\widehat{L} = \{\langle \mathsf{R}, \mathsf{G} \rangle \mid \forall \alpha \in \mathit{lexecs}(A, L) : \alpha \models \langle \mathsf{R}, \mathsf{G} \rangle\}$.

$\widehat{L}$ is the set of complemented-pairs over A which are "semantically entailed" by the complemented-pairs in L, with respect to the executions of A. In general, $\widehat{L} - L$

is nonempty. Every pair in $\widehat{L} - L$ represents a "derived" liveness property, since it is not directly specified by L, but nevertheless can be deduced from the pairs in L, when considering only the executions of A.

Definition 9 (Derived Pair) Let (A, L) be a live automaton, and let $p \in \widehat{L} - L$. Then p is a *derived pair* of (A, L).

Proposition 1 $L \subseteq \widehat{L}$.

Proof Let p be any complemented-pair in L. Hence, by definition of $\mathit{lexecs}(A, L)$, we have $\forall \alpha \in \mathit{lexecs}(A, L) : \alpha \models p$. Hence $p \in \widehat{L}$.

Proposition 2 $\mathit{lexecs}(A, \widehat{L}) = \mathit{lexecs}(A, L)$.

Proof $\mathit{lexecs}(A, \widehat{L}) \subseteq \mathit{lexecs}(A, L)$ follows immediately from Proposition 1 and the relevant definitions. Suppose $\alpha \in \mathit{lexecs}(A, L)$. By Definition 8, $\forall p \in \widehat{L} : \alpha \models p$. Hence, $\alpha \in \mathit{lexecs}(A, \widehat{L})$. Hence $\mathit{lexecs}(A, L) \subseteq \mathit{lexecs}(A, \widehat{L})$.

From Proposition 2, it follows that $(A, \widehat{L})$ is a live automaton.

4 Refining Liveness Properties Across Levels of Abstraction: Liveness-preserving Simulation Relations

The simulation relations given in Section 2.2 induce a relationship between the concrete automaton A and abstract automaton B whereby for every execution α of A there exists a corresponding, in the sense of Definition 2, execution α' of B. This correspondence between executions does not however take liveness into account. So, if we were dealing with live automata (A, L) and (B, M) instead of automata A and B, then it would be possible to have $\alpha \in \mathit{lexecs}(A, L)$, $\alpha' \notin \mathit{lexecs}(B, M)$, and $(\alpha, \alpha') \in S$ where S is a simulation relation from A to B. So, $\beta \in \mathit{traces}(\mathit{lexecs}(A, L))$ and $\beta \notin \mathit{traces}(\mathit{lexecs}(B, M))$, where $\beta = \mathit{trace}(\alpha)$, is possible. Hence establishing $A \leq_X B$ via S, where $X \in \{F, iB\}$ does not allow one to conclude $\mathit{traces}(\mathit{lexecs}(A, L)) \subseteq \mathit{traces}(\mathit{lexecs}(B, M))$, as desired, whereas it does allow one to conclude $\mathit{traces}(A) \subseteq \mathit{traces}(B)$, [18, Lemma 6.16]. For example, consider Figures 1 and 2 which respectively give a specification and a "first level" refinement of the specification, for a toy database system. The database takes input requests of the form $\mathsf{request}(x)$, where x is a query, computes a response for x using a function val (which presumably also refers to the underlying database state, we do not model this to keep the example simple), and outputs a response (x, v) where $v = \mathit{val}(x)$. This behavior is dictated by the specification in Figure 1, where received queries are placed in the set *requested*, and queries responded to are placed in the set *responded* (this prevents multiple responses to the same query). The first-level refinement of the specification (Figure 2) is identical to the specification except that it can "lose" pending requests: the $\mathsf{request}(x)$ nondeterministically chooses between adding x to *requested*, or doing nothing, as represented by $[]\mathit{skip}$ in Figure 2. Despite this fault, it is possible to establish a forward simulation F from *DB-Imp* to *DB-Spec*, as follows. A state s of *DB-Imp* and a state u of *DB-Spec* are related by F if and only if $s.\mathit{requested} \subseteq u.\mathit{requested}$ and $s.\mathit{responded} = u.\mathit{responded}$ (where $s.\mathit{var}$ denotes the value of variable var in state s). Now suppose we add the following liveness condition to both *DB-Imp* and of *DB-Spec*: $\{\langle\!\langle x \in \mathit{requested}, x \in \mathit{responded} \rangle\!\rangle \mid x \text{ is a query}\}$.

Thus, an operation x that has been requested must eventually be responded to, since $x \in \mathit{requested}$ is stable; once true, it is always true, and therefore it is true infinitely often. Now let α, α' be executions of *DB-Imp*, *DB-Spec*, respectively, which are related by F in the sense of Definition 2. Suppose some query x_0 is lost along α, and no other query is lost. Let α be live, i.e., if a query is placed into *requested*, and is not lost, then it will eventually be responded to. We now see that α' cannot be live, since $x_0 \in \mathit{requested}$ holds along an infinite suffix of α', but $x_0 \in \mathit{responded}$ never holds along α'. Hence, establishing a forward simulation from *DB-Imp* to *DB-Spec* is not sufficient to establish live trace inclusion from *DB-Imp* to *DB-Spec*.

Automaton *DB-Spec*

Signature

External:
    request(x), where x is a query
    response(x, v), where x is a query and v is a value

State

requested, a set of received queries, initially empty
responded, a set of computed responses to queries, initially empty

Actions

External request(x)	**External** response(x, v)
Pre: *true*	Pre: $x \in \mathit{requested} - \mathit{responded} \wedge v = \mathit{val}(x)$
Eff: $\mathit{requested} \leftarrow \mathit{requested} \cup \{x\}$	Eff: $\mathit{responded} \leftarrow \mathit{responded} \cup \{x\}$

Fig. 1 Specification of a simple database system

Automaton *DB-Imp*

Signature

External:
    request(x), where x is a query
    response(x, v), where x is a query and v is a value

State

requested, a set of received queries, initially empty
responded, a set of computed responses to queries, initially empty

Actions

External request(x)	**External** response(x, v)
Pre: *true*	Pre: $x \in \mathit{requested} - \mathit{responded} \wedge v = \mathit{val}(x)$
Eff: $(\mathit{requested} \leftarrow \mathit{requested} \cup \{x\})\ [\!]\ \mathit{skip}$	Eff: $\mathit{responded} \leftarrow \mathit{responded} \cup \{x\}$

Fig. 2 First level refinement of the specification of a simple database system

This example demonstrates that the simulation relations of Section 2.2 do not imply live trace inclusion. The problem is that these simulation relations do not reference the liveness conditions of the concrete and abstract automata. To remedy this, we augment the simulation relations so that every pair q in the abstract liveness condition M is related to a pair p in the concrete liveness condition L. The idea is that the simulation relation relates occurrences of states in $q.\mathsf{R}$, $q.\mathsf{G}$ in transitions of the abstract automaton (B, M) with occurrences of states in $p.\mathsf{R}$, $p.\mathsf{G}$ in transitions of the concrete automaton (A, L). The relationship is defined so that the augmented simulation implies that, in "corresponding" executions α of (A, L), α' of (B, M), if α satisfies p, then α' must satisfy q.

In more detail, an occurrence of a $q.\mathsf{R}$ state in an abstract (live) execution α' must be matched by at least one $p.\mathsf{R}$ state in the corresponding concrete (live) execution α, and an occurrence of a $p.\mathsf{G}$ state in α must be matched by at least one $q.\mathsf{G}$ state in α'. Thus, if $\alpha' \models \Box\Diamond q.\mathsf{R}$, then $\alpha \models \Box\Diamond p.\mathsf{R}$, and if $\alpha \models \Box\Diamond p.\mathsf{G}$, then $\alpha' \models \Box\Diamond q.\mathsf{G}$. Assuming α is live, we get $\alpha \models \Box\Diamond p.\mathsf{R} \Rightarrow \Box\Diamond p.\mathsf{G}$. This and the previous two implications yields $\alpha' \models \Box\Diamond q.\mathsf{R} \Rightarrow \Box\Diamond q.\mathsf{G}$. Hence α' is live. Hence we can show that if an abstract execution α' and concrete execution α correspond (according to the simulation), and α is live, then α' is also live. The matching thus allows us to show that every live execution of (A, L) has a "corresponding" live execution in (B, M). Live trace inclusion follows immediately.

Since the semantic closure $\widehat{L}$ of L specifies the same set of live executions (Proposition 2), as L does, we can relax the requirement $p \in L$ to $p \in \widehat{L}$. Since $\widehat{L}$ is in general a superset of L, this can be very helpful in refining the abstract liveness condition. In particular, it enables us to split the refinement task into two subtasks: refinement across abstraction levels (which we address in this section) and refinement within an abstraction level (which we address in the next section).

Let (A, L) be a live automaton, α be a finite execution fragment of A, and $p \in L$. We abuse notation and write $\alpha \in p.\mathsf{R}$ iff there exists a state s along α such that $s \in p.\mathsf{R}$. $\alpha \in p.\mathsf{G}$ is defined similarly. The above considerations lead to the following definitions of liveness-preserving simulation relations.

Definition 10 (Liveness-preserving Forward Simulation w.r.t. Invariants) Let (A, L) and (B, M) be live automata with the same external actions. Let I_A, I_B be invariants of A, B respectively. Let $f = (g, h)$ where $g \subseteq \mathit{states}(A) \times \mathit{states}(B)$ and $h : M \mapsto \widehat{L}$ is a total mapping over M[4]. Then f is a *liveness-preserving forward simulation* from (A, L) to (B, M) with respect to I_A and I_B iff:

1. If $s \in \mathit{start}(A)$, then $g[s] \cap \mathit{start}(B) \neq \emptyset$.
2. If $s \xrightarrow{a}_A s'$, $s \in I_A$, and $u \in g[s] \cap I_B$, then there exists a finite execution fragment α of B such that $\mathit{fstate}(\alpha) = u$, $\mathit{lstate}(\alpha) \in g[s']$, and $\mathit{trace}(\alpha) = \mathit{trace}(a)$. Furthermore, for all $q \in M$,
   (a) if $\alpha \in q.\mathsf{R}$ then $s \in p.\mathsf{R}$ or $s' \in p.\mathsf{R}$, and
   (b) if $s \in p.\mathsf{G}$ or $s' \in p.\mathsf{G}$ then $\alpha \in q.\mathsf{G}$,
   where $p = h(q)$.
3. Call a transition $s \xrightarrow{a}_A s'$ *always-silent* iff $s \in I_A$ and for every finite execution fragment α of B such that $\mathit{fstate}(\alpha) \in g[s] \cap I_B$, $\mathit{lstate}(\alpha) \in g[s']$, and $\mathit{trace}(\alpha) = \mathit{trace}(a)$, we have $|\alpha| = 0$, i.e., α consists of a single state. In other words, the transition $s \xrightarrow{a}_A s'$ is matched only by the empty transition in B. Then, g is such

[4] That is, $h(q)$ is defined for all $q \in M$.

that every live execution of (A, L) contains an infinite number of transitions that are not always-silent.

Clause 1 is the usual condition of a forward simulation requiring that every start state of (A, L) be related to at least one start state of (B, M).

Clause 2 is the condition of a forward simulation which requires that every transition $s \xrightarrow{a}_A s'$ of (A, L) be "simulated" by an execution fragment α of (B, M) which has the same trace. The role of the invariants I_A and I_B is to weaken this clause to apply only to a superset of the reachable states, rather than to all states of automata A, B, respectively, We also require that every complemented-pair $q \in M$ is matched to a complemented-pair $p \in \widehat{L}$ by the mapping h and that such corresponding pairs impose a constraint on the transition $s \xrightarrow{a}_A s'$ of (A, L) and the simulating execution fragment α of (B, M), as follows. If α contains some $q.\mathsf{R}$ state, then at least one of s, s' is a $p.\mathsf{R}$ state, and if at least one of s, s' is a $p.\mathsf{G}$ state, then α contains some $q.\mathsf{G}$ state. This requirement thus enforces the matching discussed at the beginning of this section, from which live trace inclusion follows.

Clause 3 is needed to ensure that a live execution of (A, L) has at least one corresponding *infinite* execution in (B, M). This execution can then be shown, using clause 2, to be live (see Lemma 5 below). If $s \xrightarrow{a}_A s'$ is always-silent, then a must be an internal action. Thus, in practice, clause 3 holds, since executions with an (infinite) suffix consisting solely of internal actions are not usually considered to be live. Clause 3 can itself be expressed as a complemented-pair (which is added to L). Call an action a of A *non-always-silent* iff no transition arising from its execution is always-silent. Thus, every transition arising from the execution of a can be matched with respect to g by some nonempty execution fragment of B. It is also possible that the transition can be matched by the empty fragment, but what is important is that it is always possible to *choose* a nonempty fragment to match with. This means that we can always match a live execution α of (A, L) with some *infinite* execution of (B, M), by always matching the non-always-silent transitions in α with nonempty execution fragments of (B, M).

By definition, any external action of A is non-always-silent. An internal action of A may or may not be non-always-silent. We introduce an auxiliary boolean variable *nonalwayssilent* that is set to *true* each time a non-always-silent action of A is executed, and is set to *false* infinitely often by a new internal action of A whose precondition is *true* and whose effect is *nonalwayssilent* := *false* (every execution of this new action can be simulated by the empty transition in B, since *nonalwayssilent* has no effect on any other state component of A, nor on the execution of other actions in A). Then the pair $\langle\!\langle \mathit{true}, \mathit{nonalwayssilent} \rangle\!\rangle$ expresses that a non-always-silent action of A is executed infinitely often, which implies that each live execution of (A, L) contains an infinite number of non-always-silent transitions. The pair $\langle\!\langle \mathit{true}, \mathit{nonalwayssilent} \rangle\!\rangle$ can then be refined at the next lower level of abstraction in exactly the same way as all the other pairs in L. See Section 6 for an example of this technique.

It is clear from the definitions that if (g, h) is a liveness-preserving forward simulation from (A, L) to (B, M) w.r.t. invariants, then g is a forward simulation from A to B w.r.t. the same invariants. We write $(A, L) \leq_{\ell F} (B, M)$ if there exists a liveness-preserving forward simulation from (A, L) to (B, M) w.r.t. invariants, and $(A, L) \leq_{\ell F} (B, M)$ via f if f is a liveness-preserving forward simulation from (A, L) to (B, M) w.r.t. invariants.

Definition 11 (Liveness-preserving Backward Simulation w.r.t. Invariants) Let (A, L) and (B, M) be live automata with the same external actions. Let I_A, I_B

be invariants of A, B respectively. Let $b = (g, h)$ where $g \subseteq \mathit{states}(A) \times \mathit{states}(B)$ and $h : M \mapsto \widehat{L}$ is a total mapping over M. Then b is a *liveness-preserving backward simulation* from (A, L) to (B, M) with respect to I_A and I_B iff:

1. If $s \in I_A$, then $g[s] \cap I_B \neq \emptyset$.
2. If $s \in \mathit{start}(A)$, then $g[s] \cap I_B \subseteq \mathit{start}(B)$.
3. If $s \xrightarrow{a}_A s'$, $s \in I_A$, and $u' \in g[s'] \cap I_B$, then there exists a finite execution fragment α of B such that $\mathit{fstate}(\alpha) \in g[s] \cap I_B$, $\mathit{lstate}(\alpha) = u'$, and $\mathit{trace}(\alpha) = \mathit{trace}(a)$. Furthermore, for all $q \in M$,
   (a) if $\alpha \in q.\mathsf{R}$ then $s \in p.\mathsf{R}$ or $s' \in p.\mathsf{R}$, and
   (b) if $s \in p.\mathsf{G}$ or $s' \in p.\mathsf{G}$ then $\alpha \in q.\mathsf{G}$,
   where $p = h(q)$.
4. Call a transition $s \xrightarrow{a}_A s'$ *sometimes-silent* iff $s \in I_A$ and for some finite execution fragment α of B such that $\mathit{fstate}(\alpha) \in g[s] \cap I_B$, $\mathit{lstate}(\alpha) \in g[s']$, and $\mathit{trace}(\alpha) = \mathit{trace}(a)$, we have $|\alpha| = 0$, i.e., α consists of a single state. In other words, the transition $s \xrightarrow{a}_A s'$ can be matched by the empty transition in B. Then, g is such that every live execution of (A, L) contains an infinite number of transitions that are not sometimes-silent.

Clauses 1 and 2 are the usual conditions of a backward simulation requiring that a state in the invariant I_A of (A, L) is related to at least one state in the invariant I_B of (B, M), and that every start state of (A, L) is related only to start states of (B, M), ignoring states not in the invariant I_B. These clauses are needed due to the "backwards" nature of the bisimulation, since, from a state u' in the invariant I_B it is possible, when "going backwards" along a transition to reach a state u not in the invariant, i.e., $u \xrightarrow{a}_B u'$, $u \notin I_B$, and $u' \in I_B$ is possible. Also, a start state in (A, L) must *always* be matched by a start state in (B, M), since the matching state in (B, M) cannot be chosen initially: it is constrained by the succeeding transitions, i.e., it is "chosen" last of all, and so the result must be an initial state of B regardless of the choice.

Clause 3 is the condition of a backward simulation which requires that every transition $s \xrightarrow{a}_A s'$ of (A, L) be "simulated" by an execution fragment α of (B, M), except that we also require that every complemented-pair $q \in M$ is matched to a complemented-pair $p \in \widehat{L}$ by the mapping h and that such corresponding pairs impose a constraint on the transition $s \xrightarrow{a}_A s'$ of (A, L) and the simulating execution fragment α of (B, M), as follows. If α contains some $q.\mathsf{R}$ state, then at least one of s, s' is a $p.\mathsf{R}$ state, and if at least one of s, s' is a $p.\mathsf{G}$ state, then α contains some $q.\mathsf{G}$ state. This requirement thus enforces the matching discussed at the beginning of this section, from which live trace inclusion follows.

Clause 4 is needed to ensure that a live execution of (A, L) has at least one corresponding *infinite* execution in (B, M). This execution can then be shown, using clause 3, to be live (see Lemma 5 below). If $s \xrightarrow{a}_A s'$ is sometimes-silent, then a must be an internal action. Thus, in practice, clause 4 holds, since executions with an (infinite) suffix consisting solely of internal actions are not usually considered to be live. Clause 4 can itself be expressed as a complemented-pair (which is added to L), which can then be refined at the next lower level of abstraction. Call an action a of A *non-sometimes-silent* iff no transition arising from its execution is sometimes-silent. Thus, every transition arising from the execution of a must *always* be matched with respect to g by some nonempty execution fragment of B.

We can now express the requirement that a non-sometimes-silent action of A is executed infinitely often, as a complemented-pair, and refine this pair at the next lower level of abstraction. The details are similar to those discussed above for Clause 3 of Definition 10, and are omitted. Note the difference with forward simulation; there, we only had to ensure that it was infinitely often possible to *choose* a nonempty execution fragment to match with. With backward simulations, we have to show that infinitely often, *all* the matching execution fragments are nonempty.

It is clear from the definitions that if (g, h) is a liveness-preserving backward simulation from (A, L) to (B, M) w.r.t. some invariants, then g is a backward simulation from A to B w.r.t. the same invariants. We write $A \leq_{\ell B} B$ if there exists a liveness-preserving backward simulation from A to B w.r.t. some invariants, and $A \leq_{\ell B} B$ via b if b is a liveness-preserving backward simulation from A to B w.r.t. some invariants. If the backward simulation g is image-finite, then we write $A \leq_{i\ell B} B$, $A \leq_{i\ell B} B$ via b, respectively.

We use ℓF to denote liveness-preserving forward simulation and $i\ell B$ to denote image-finite liveness-preserving backward simulation. We write $X \in \{\ell F, i\ell B\}$ to mean that X is one of these relations.

Liveness-preserving simulation relations induce a correspondence between the live executions of the concrete and the abstract automata. This correspondence is captured by the notion of R^ℓ-relation. We remind the reader of the definition $\mathit{trace}(\alpha', j, k) = \mathit{trace}(b_j \cdots b_k)$ if $j \leq k$, and $= \lambda$ (the empty sequence) if $j > k$.

Definition 12 (R^ℓ-relation and Live Index Mappings) Let (A, L) and (B, M) be live automata with the same external actions. Let $R^\ell = (R, H)$ where R is a relation over $\mathit{states}(A) \times \mathit{states}(B)$ and $H : M \mapsto \widehat{L}$ is a total mapping over M. Furthermore, let α and α' be executions of (A, L) and (B, M), respectively:

$\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$

$\alpha' = u_0 b_1 u_1 b_2 u_2 \cdots$

Say that α and α' are *R^ℓ-related*, written $(\alpha, \alpha') \in R^\ell$, if there exists a total, nondecreasing mapping $m : \{0, 1, \ldots, |\alpha|\} \mapsto \{0, 1, \ldots, |\alpha'|\}$ such that:

1. $m(0) = 0$,
2. $(s_i, u_{m(i)}) \in R$ for all i, $0 \leq i \leq |\alpha|$,
3. $\mathit{trace}(\alpha', m(i-1)+1, m(i)) = \mathit{trace}(a_i)$ for all i, $0 < i \leq |\alpha|$,
4. for all $j, 0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$, and
5. for all complemented-pairs $q \in M$ and all i, $0 < i \leq |\alpha|$:
   (a) if $(\exists j \in m(i-1) \ldots m(i) : u_j \in q.\mathsf{R})$ then $s_{i-1} \in p.\mathsf{R}$ or $s_i \in p.\mathsf{R}$, and
   (b) if $s_{i-1} \in p.\mathsf{G}$ or $s_i \in p.\mathsf{G}$ then $(\exists j \in m(i-1) \ldots m(i) : u_j \in q.\mathsf{G})$,
   where $p = H(q)$.

The mapping m is referred to as a *live index mapping* from α to α' with respect to R^ℓ. Write $((A, L), (B, M)) \in R^\ell$ if for every live execution α of (A, L), there exists a live execution α' of (B, M) such that $(\alpha, \alpha') \in R^\ell$.

Note that $(\alpha, \alpha') \in R^\ell$ does not require α, α' to be live executions. By Definitions 2 and 12, it is clear that, if $R^\ell = (R, H)$, then $(\alpha, \alpha') \in R^\ell$ implies $(\alpha, \alpha') \in R$. The following lemma establishes a correspondence between the prefixes of a live execution of the concrete automaton and an infinite family of finite executions of the abstract automaton.

Lemma 2 *Let (A, L) and (B, M) be live automata with the same external actions, and such that $(A, L) \leq_{\ell F} (B, M)$ via f for some $f = (g, h)$. Let α be an arbitrary live execution of (A, L). Then there exists a collection $(\alpha'_i, m_i)_{0 \leq i}$ of finite executions of (B, M) and mappings such that:*

1. *m_i is a live index mapping from $\alpha|_i$ to α'_i with respect to f, for all $i \geq 0$, and*
2. *$\alpha'_{i-1} \leq \alpha'_i$ and $m_{i-1} = m_i \lceil \{0, \ldots, i-1\}$ for all $i > 0$, and*
3. *$\alpha'_{i-1} < \alpha'_i$ for infinitely many $i > 0$.*

Proof Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ and let I_A, I_B be invariants of A, B, respectively, such that f is a liveness-preserving forward simulation from (A, L) to (B, M) with respect to I_A and I_B. We construct α'_i and m_i by induction on i.

Since $s_0 \in \mathit{start}(A)$, we have $(s_0, v_0) \in g$ and $v_0 \in \mathit{start}(B)$ for some state v_0, by Definition 10, clause 1. Let $\alpha'_0 = v_0$ and let m_0 be the mapping that maps 0 to 0. Then, m_0 is a live index mapping from $\alpha|_0$ to α'_0 with respect to f (in particular, clause 5 of Definition 12 holds vacuously, since $|\alpha|_0| = 0$).

Now inductively assume that m_{i-1} (for $i > 0$) is a live index mapping from $\alpha|_{i-1}$ to α'_{i-1} with respect to f. Let $u_0 = \mathit{lstate}(\alpha'_{i-1})$. Then, by clause 4 of Definition 12 and the fact that m_{i-1} is nondecreasing, we have $m_{i-1}(i-1) = |\alpha'_{i-1}|$ and $(s_{i-1}, u_0) \in g$. Since s_{i-1}, s_i, and u_0 are reachable, by definition, they satisfy their respective invariants. Hence, by Definition 10, clause 2, there exists a finite execution fragment $u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B \cdots \xrightarrow{b_n}_B u_n$ of B such that $u_n \in g[s_i]$, $\mathit{trace}(b_1 \cdots b_n) = \mathit{trace}(a_i)$, and for all complemented-pairs $q \in M$:

1. if $(\exists j \in 1 \ldots n : u_j \in q.\mathsf{R})$ then $s_{i-1} \in p.\mathsf{R}$ or $s_i \in p.\mathsf{R}$, and
2. if $s_{i-1} \in p.\mathsf{G}$ or $s_i \in p.\mathsf{G}$ then $(\exists j \in 1 \ldots n : u_j \in q.\mathsf{G})$,

where $p = h(q)$. Now define $\alpha'_i = \alpha'_{i-1} {}^\frown (u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B \cdots \xrightarrow{b_n}_B u_n)$, and define m_i to be the mapping such that $m_i(j) = m_{i-1}(j)$ for all j, $0 \leq j \leq i-1$, and $m_i(i) = |\alpha'_i|$. We argue that m_i is a live index mapping from $\alpha|_i$ to α'_i with respect to f, i.e., that all clauses of Definition 12 hold. Clause 1 holds since $m_i(0) = m_{i-1}(0)$ by definition, and $m_{i-1}(0) = 0$ by the inductive hypothesis. Clause 2 holds by the inductive hypothesis and $u_n \in g[s_i]$. Clause 3 holds by the inductive hypothesis and $\mathit{trace}(b_1 \cdots b_n) = \mathit{trace}(a_i)$. Clause 4 holds since $m_i(|\,\alpha|_i\,|) = m_i(i) = |\alpha'_i|$, by definition. Finally, clause 5 holds by the inductive hypothesis and the conditions for all complemented-pairs $q \in M$ just established above w.r.t. $s_{i-1} \xrightarrow{a_i}_A s_i$ and $u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B \cdots \xrightarrow{b_n}_B u_n$. Having established that m_i is a live index mapping from $\alpha|_i$ to α'_i with respect to f, we conclude that clause 1 of the lemma holds.

Clause 2 of the lemma holds by construction of α'_i and m_i, since α'_i and m_i are obtained by extending α'_{i-1} and m_{i-1}, respectively.

By Definition 10, clause 3, for infinitely many $i > 0$, we can select the execution fragment $u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B \cdots \xrightarrow{b_n}_B u_n$ that matches $s_{i-1} \xrightarrow{a_i}_A s_i$ so that $n > 0$. Hence, for infinitely many $i > 0$, we have $\alpha'_{i-1} < \alpha'_i$. Thus, clause 3 of the lemma holds.

Definition 13 (Induced Digraph) Let (A, L) and (B, M) be live automata with the same external actions and assume $A \leq_{i\ell B} B$ via $b = (g, h)$ with respect to invariants I_A and I_B. For any execution $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ of A, let the *digraph induced* by α, b, I_B, L, and M be the directed graph G given as follows:

1. The nodes of G are the ordered pairs (u, i) such that $0 \leq i \leq |\alpha|$, and $u \in g[s_i] \cap I_B$, and
2. there is an edge from (u, i) to (u', i') iff $i' = i + 1$ and there exists a finite execution fragment α' of B such that $\mathit{fstate}(\alpha') = u$, $\mathit{lstate}(\alpha') = u'$, $\mathit{trace}(\alpha') = \mathit{trace}(a_{i+1})$, and for all complemented-pairs $q \in M$:
   (a) if $\alpha' \in q.\mathsf{R}$ then $s_i \in p.\mathsf{R}$ or $s_{i+1} \in p.\mathsf{R}$, and
   (b) if $s_i \in p.\mathsf{G}$ or $s_{i+1} \in p.\mathsf{G}$ then $\alpha' \in q.\mathsf{G}$,
   where $p = h(q)$.

Lemma 3 *Let (A, L) and (B, M) be live automata with the same external actions and assume $A \leq_{i\ell B} B$ via b with respect to invariants I_A and I_B. Let α be any execution of A. Then the digraph G induced by α, b, I_B, L, and M satisfies:*

1. *For each i, $0 \leq i \leq |\alpha|$, there is at least one node in G of the form (u, i).*
2. *The roots of G are exactly the nodes of the form $(u, 0)$.*
3. *G has a finite number of roots.*
4. *Each node in G has finite outdegree.*
5. *Each node of G is reachable from some root of G.*

Proof Let $b = (g, h)$. Then g is an image-finite backward simulation from A to B. We deal with each clause in turn.

1. Each state s_i of α is reachable, and so belongs to I_A. Hence $g[s_i] \cap I_B \neq \emptyset$ by Clause 1 of Definition 11. Hence by Definition 13, clause 1, there exist nodes of G of the form (u, i).
2. Every node $(u, 0)$ is a root of G (i.e., it has no incoming edges). We now show that any node (u, i) with $i > 0$ cannot be a root. Now $u \in g[s_i] \cap I_B$ by Definition 13, clause 1. Also, $s_{i-1} \in I_A$ and $s_{i-1} \xrightarrow{a_i}_A s_i$ by assumption, hence by Definition 11, clause 3, there exists a finite execution fragment α' of B such that $\mathit{fstate}(\alpha') \in g[s_{i-1}] \cap I_B$, $\mathit{lstate}(\alpha') = u$, $\mathit{trace}(\alpha') = \mathit{trace}(a_i)$, and, for all $q \in M$,
   (a) if $\alpha' \in q.\mathsf{R}$ then $s_{i-1} \in p.\mathsf{R}$ or $s_i \in p.\mathsf{R}$, and
   (b) if $s_{i-1} \in p.\mathsf{G}$ or $s_i \in p.\mathsf{G}$ then $\alpha' \in q.\mathsf{G}$,
   where $p = h(q)$. Hence, by Definition 13, clause 2, there exists an edge in G from $(\mathit{fstate}(\alpha'), i - 1)$ to (u, i).
3. Since g is image-finite, the set $g[s_0] \cap I_B$ is finite. By Definition 13, clause 1, all nodes of G of the form $(u, 0)$ must satisfy $u \in g[s_0] \cap I_B$. Hence, there are a finite number of such nodes. By clause 2 of the lemma (which has already been established), these nodes are exactly the roots of G. Hence, the number of roots is finite.
4. Let (u, i) be an arbitrary node of G. By Definition 13, clause 2, from any node of the form (u, i), all outgoing edges are to nodes of the form $(u', i + 1)$. Since g is image-finite, the set $g[s_{i+1}] \cap I_B$ is finite. By Definition 13, clause 1, all nodes of G of the form $(u, i+1)$ must satisfy $u \in g[s_{i+1}] \cap I_B$. Hence, there are a finite number of such nodes. Hence, the outdegree of any node of G of the form (u, i) is finite. Since (u, i) was chosen arbitrarily, the result follows.
5. We establish this by induction on the second component i of the nodes (u, i) of G. For the base case, $i = 0$ and nodes $(u, 0)$ are reachable by definition since they are roots. Assume the induction hypothesis that all nodes of the form (u, i) are reachable from some root of G, and consider an arbitrary node of the form $(u, i + 1)$.

   Now $u \in g[s_{i+1}] \cap I_B$ by Definition 13, clause 1. Also, $s_i \in I_A$ and $s_i \xrightarrow{a_{i+1}}_A s_{i+1}$ by assumption, hence by Definition 11, clause 3, there exists a finite execution fragment

α' of B such that $fstate(\alpha') \in g[s_i] \cap I_B$, $lstate(\alpha') = u$, $trace(\alpha') = trace(a_{i+1})$, and, for all $q \in M$,
(a) if $\alpha' \in q.\mathsf{R}$ then $s_i \in p.\mathsf{R}$ or $s_{i+1} \in p.\mathsf{R}$, and
(b) if $s_i \in p.\mathsf{G}$ or $s_{i+1} \in p.\mathsf{G}$ then $\alpha' \in q.\mathsf{G}$,
where $p = h(q)$. Hence, by Definition 13, clause 2, there exists an edge in G from $(fstate(\alpha'), i)$ to $(u, i+1)$. By the induction hypothesis, $(fstate(\alpha'), i)$ is reachable. Hence, so is $(u, i+1)$.

Since all the clauses are established, Lemma 3 holds.

Lemma 4 *Let (A, L) and (B, M) be live automata with the same external actions, and such that $(A, L) \leq_{i\ell B} (B, M)$ via b for some $b = (g, h)$. Let α be an arbitrary live execution of (A, L). Then there exists a collection $(\alpha'_i, m_i)_{0 \leq i}$ of finite executions of (B, M) and mappings such that:*

1. *m_i is a live index mapping from $\alpha|_i$ to α'_i with respect to b, for all $i \geq 0$, and*
2. *$\alpha'_{i-1} \leq \alpha'_i$ and $m_{i-1} = m_i \lceil \{0, \ldots, i-1\}$ for all $i > 0$, and*
3. *$\alpha'_{i-1} < \alpha'_i$ for infinitely many $i > 0$.*

Proof Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ and let I_A, I_B be invariants of A, B, respectively, such that b is a image-finite liveness-preserving backward simulation from (A, L) to (B, M) with respect to I_A and I_B. Let G be the digraph induced by α, b, I_B, L and M. Since α is infinite (all live executions are infinite, by Definition 5), G is infinite. Hence, by clauses 3 and 4 of Lemma 3, and Konig's lemma, G contains an infinite path. Fix $p = (u_0, 0)(u_1, 1), \ldots$ to be any such path. By Definition 13, clause 1, $u_i \in g[s_i] \cap I_B$ for all $i \geq 0$. We now construct α'_i and m_i by induction on i, with α'_i such that $lstate(\alpha'_i) = u_i$.

Now $s_0 \in start(A)$ since α is an execution of A. Also, by Definition 13, $u_0 \in g[s_0] \cap I_B$. Hence, by clause 2 of Definition 11, $u_0 \in start(B)$. Let $\alpha'_0 = u_0$ and let m_0 be the mapping that maps 0 to 0. Then, m_0 is a live index mapping from $\alpha|_0$ to α'_0 with respect to b (in particular, clause 5 of Definition 12 holds vacuously, since $|\alpha|_0| = 0$), and $lstate(\alpha'_0) = u_0$.

Now inductively assume that m_{i-1} (for $i > 0$) is a live index mapping from $\alpha|_{i-1}$ to α'_{i-1} with respect to b, and that $lstate(\alpha'_{i-1}) = u_{i-1}$. By construction of path p, there is an edge in G from $(u_{i-1}, i-1)$ to (u_i, i). Hence, by Definition 13, there exists a finite execution fragment α'' such that $fstate(\alpha'') = u_{i-1}$, $lstate(\alpha'') = u_i$, $trace(\alpha'') = trace(a_i)$, and, for all complemented-pairs $q \in M$:

1. if $\alpha'' \in q.\mathsf{R}$ then $s_{i-1} \in p.\mathsf{R}$ or $s_i \in p.\mathsf{R}$, and
2. if $s_{i-1} \in p.\mathsf{G}$ or $s_i \in p.\mathsf{G}$ then $\alpha'' \in q.\mathsf{G}$,

where $p = h(q)$. Now define $\alpha'_i = \alpha'_{i-1} {}^\frown \alpha''$, and define m_i to be the mapping such that $m_i(j) = m_{i-1}(j)$ for all j, $0 \leq j \leq i-1$, and $m_i(i) = |\alpha'_i|$. We argue that m_i is a live index mapping from $\alpha|_i$ to α'_i with respect to b, i.e., that all clauses of Definition 12 hold, and that $lstate(\alpha'_i) = u_i$. Clause 1 holds since $m_i(0) = m_{i-1}(0)$ by definition, and $m_{i-1}(0) = 0$ by the inductive hypothesis. Clause 2 holds by the inductive hypothesis, $lstate(\alpha'') = u_i$, and $u_i \in g[s_i]$ (which we established above). Clause 3 holds by the inductive hypothesis and $trace(\alpha'') = trace(a_i)$. Clause 4 holds since $m_i(|\,\alpha|_i\,|) = m_i(i) = |\alpha'_i|$, by definition. Finally, clause 5 holds by the inductive hypothesis and the conditions for all complemented-pairs $q \in M$ established above w.r.t. $s_{i-1} \xrightarrow{a_i}_A s_i$ and α''. Having established that m_i is a live index mapping from

$\alpha|_i$ to α'_i with respect to f, we conclude that clause 1 of the lemma holds. Also, $lstate(\alpha'_i) = lstate(\alpha'') = u_i$, as required for the induction step to be valid.

Clause 2 of the lemma holds by construction of α'_i and m_i, since α'_i and m_i are obtained by extending α'_{i-1} and m_{i-1}, respectively.

By Definition 11, clause 4, for infinitely many $i > 0$, the execution fragment α'' which matches $s_{i-1} \xrightarrow{a_i}_A s_i$ must have length $|\alpha''| \geq 1$. Hence, for infinitely many $i > 0$, we have $\alpha'_{i-1} < \alpha'_i$. Thus, clause 3 of the lemma holds.

Our next lemma shows that, if infinite concrete and abstract executions correspond in the sense of $(\alpha, \alpha') \in R^\ell$, and the concrete execution is live, then so is the abstract execution.

Lemma 5 *Let (A, L) and (B, M) be live automata with the same external actions. Let $R^\ell = (R, H)$ where R is a relation over $states(A) \times states(B)$ and $H : M \mapsto \widehat{L}$ is a total mapping over M. Let α, α' be arbitrary infinite executions of (A, L), (B, M) respectively. If $(\alpha, \alpha') \in R^\ell$, then $\alpha \in lexecs(A, L)$ implies $\alpha' \in lexecs(B, M)$.*

Proof We assume the antecedents of the lemma and establish $\alpha' \notin lexecs(B, M)$ implies $\alpha \notin lexecs(A, L)$. Let:

$\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$

$\alpha' = u_0 b_1 u_1 b_2 u_2 \cdots$

Since $(\alpha, \alpha') \in R^\ell$, there exists a live index mapping $m : \{0, 1, \ldots, |\alpha|\} \mapsto \{0, 1, \ldots, |\alpha'|\}$ satisfying the conditions in Definition 12. Suppose $\alpha' \notin lexecs(B, M)$. Then, by Definition 6, there exists a complemented-pair $q \in M$ such that $\alpha' \models \Box\Diamond q.\mathsf{R} \wedge \Diamond\Box\neg q.\mathsf{G}$. Let $p = H(q)$. We prove:

$$\alpha \models \Box\Diamond p.\mathsf{R} \wedge \Diamond\Box\neg p.\mathsf{G}. \qquad (*)$$

Since $\alpha' \models \Box\Diamond q.\mathsf{R}$, there exist an infinite number of pairs of states $(u_{m(i-1)}, u_{m(i)})$ along α' that contain a $q.\mathsf{R}$-state between them (inclusive, i.e., the $q.\mathsf{R}$-state could be $u_{m(i-1)}$ or $u_{m(i)}$). By clauses 2 and 3 of Definition 12, for each such pair there corresponds a pair of states (s_{i-1}, s_i) along α such that $(s_{i-1}, u_{m(i-1)}) \in R$ and $(s_i, u_{m(i)}) \in R$. Also, by clause 5a of Definition 12, $s_{i-1} \in p.\mathsf{R}$ or $s_i \in p.\mathsf{R}$. Since this holds for an infinite number of values of the index i, we conclude

$$\alpha \models \Box\Diamond p.\mathsf{R}. \qquad (a)$$

Since $\alpha' \models \Diamond\Box\neg q.\mathsf{G}$, there exists a state u_g along α' such that $\forall \ell \geq g : u_\ell \notin q.\mathsf{G}$. Now assume that $\alpha \models \Box\Diamond p.\mathsf{G}$. Since m is nondecreasing and cofinal in $\{0, 1, \ldots, |\alpha'|\}$ (clause 4, Definition 12), there exists an s_{i-1} along α such that $s_{i-1} \in p.\mathsf{G}$ and $m(i-1) \geq g$. By clauses 2 and 3 of Definition 12, $(s_{i-1}, u_{m(i-1)}) \in R$ and $(s_i, u_{m(i)}) \in R$. Also, by clause 5b of Definition 12, at least one of $u_{m(i-1)}, u_{m(i-1)+1}, \ldots, u_{m(i)}$ is a $q.\mathsf{G}$ state. Since $m(i-1) \geq g$, this contradicts $\forall \ell \geq g : u_\ell \notin q.\mathsf{G}$ above. Hence the assumption $\alpha \models \Box\Diamond p.\mathsf{G}$ must be false, and so:

$$\alpha \models \Diamond\Box\neg p.\mathsf{G}. \qquad (b)$$

From (a) and (b), we conclude (*). From (*), we have $\alpha \not\models p$. Now $p \in \widehat{L}$, since $H : M \mapsto \widehat{L}$. Hence, $\alpha \notin lexecs(A, \widehat{L})$ by Definition 6. Hence, by Proposition 2, $\alpha \notin lexecs(A, L)$.

We can now establish a correspondence theorem for live executions. Our theorem states that, if a liveness-preserving simulation relation S^ℓ is established from a concrete automaton to an abstract automaton, then for every live execution α of the concrete automaton, there exists a corresponding (in the sense of $(\alpha, \alpha') \in S^\ell$) live execution α'

of the abstract automaton. Our proof uses Lemmas 2 and 4 to establish the existence of an infinite family of finite executions corresponding to prefixes of α. We then construct α' from this infinite family using the "diagonalization" technique of [18]. Finally, we invoke Lemma 5 to show that α' is live, given that α is live.

Theorem 2 (Live Execution Correspondence Theorem) *Let (A, L) and (B, M) be live automata with the same external actions. Suppose $(A, L) \leq_X (B, M)$ via S^ℓ, where $X \in \{\ell F, i\ell B\}$. Then $((A, L), (B, M)) \in S^\ell$.*

Proof We proceed by cases on X.

Case 1 $X = \ell F$. So S^ℓ is a liveness-preserving forward simulation $f = (g, h)$, and $(A, L) \leq_{\ell F} (B, M)$ via f. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an arbitrary live execution of (A, L), and let $(\alpha'_i, m_i)_{0 \leq i}$ be a collection of finite executions of (B, M) and mappings as given by Lemma 2. By definition of $((A, L), (B, M)) \in f$, we must show that there exists a live execution α' of (B, M) such that $(\alpha, \alpha') \in f$.

By Definition 6, α is infinite. Let m be the unique mapping over the natural numbers defined by $m(i) = m_i(i)$, for all $i \geq 0$. Let α' be the limit of α'_i under the prefix ordering, that is, α' is the unique execution of (B, M) defined by $\alpha'|_{m(i)} = \alpha'_i$ for all $i \geq 0$, with the restriction that for any index j of α', there exists an i such that $\alpha'|_j \leq \alpha'_i$. By Lemma 2, clause 3, α' is infinite.

We now show that m is a live index mapping from α to α' with respect to f. The proof that m is nondecreasing and total and satisfies clauses 1–4 of Definition 12 proceeds in exactly the same way that the proof of the corresponding assertions does in the proof of the Execution Correspondence Theorem in [18]. We repeat the details for sake of completeness.

Suppose m is not nondecreasing. Then there exists an i such that $m(i) < m(i-1)$. However, $m(i) = m_i(i)$ and $m(i-1) = m_{i-1}(i-1) = m_i(i-1)$, so this contradicts the fact that m_i is an index mapping and is therefore nondecreasing. Likewise, we can see that the range of m is within $\{0, \ldots, |\alpha'|\}$.

Clause 1 of Definition 12 holds since m_0 is an index mapping and therefore satisfies $m_0(0) = 0$. Hence $m(0) = m_0(0) = 0$. Assume clauses 2 or 3 do not hold. Then, there must exist an i for which one of the clauses is invalidated. However, this contradicts the fact that, for all i, m_i is an index mapping from $\alpha|_i$ to α'_i with respect to f. Now assume that clause 4 does not hold. Hence, there is an index j in α' such that $m(i) < j$ for all i. By definition of α', there exists an i such that $\alpha'|_j \leq \alpha'_i$. Thus $|\alpha'_i| \geq j$. Now Lemma 2 gives us $m_i(i) = |\alpha'_i|$. Hence $m(i) \geq j$, since $m(i) = m_i(i)$. This contradicts $m(i) < j$.

Now assume that m violates clause 5 of Definition 12. Then, there exists a pair $q \in M$ and an $i > 0$ for which clause 5 is invalidated. However, this contradicts the fact that, for all $i > 0$, m_i is a live index mapping from $\alpha|_i$ to α'_i with respect to f (Lemma 2, clause 1). Hence m satisfies clause 5 of Definition 12. Since m satisfies all clauses of Definition 12, m is a live index mapping from α to α' with respect to f, and so $(\alpha, \alpha') \in f$. Since $\alpha \in \mathit{lexecs}(A, L)$, $(\alpha, \alpha') \in f$, and α, α' are both infinite, we can apply Lemma 5 to conclude $\alpha' \in \mathit{lexecs}(B, M)$, i.e., α' is a live execution of (B, M), which establishes the theorem in this case.

Case 2 $X = i\ell B$. So S^ℓ is an image-finite liveness-preserving backward simulation $b = (g, h)$, and $(A, L) \leq_{i\ell B} (B, M)$ via b. The argument is identical to that of Case 1, except that we invoke Lemma 4 instead of Lemma 2.

Since both cases of X have been dealt with, the theorem is established.

We now establish our main result: liveness-preserving simulation relations imply the live preorder.

Theorem 3 (Liveness) *Let (A, L) and (B, M) be live automata with the same external actions. Suppose $(A, L) \leq_X (B, M)$, where $X \in \{\ell F, i\ell B\}$. Then $(A, L) \sqsubseteq_\ell (B, M)$.*

Proof From $(A, L) \leq_X (B, M)$, we have $(A, L) \leq_X (B, M)$ via S^ℓ for some $S^\ell = (g, h)$. We establish $\mathit{traces}(\mathit{lexecs}(A, L)) \subseteq \mathit{traces}(\mathit{lexecs}(B, M))$, which, by Definition 7, proves the theorem. Let β be an arbitrary trace in $\mathit{traces}(\mathit{lexecs}(A, L))$. By definition, $\beta = \mathit{trace}(\alpha)$ for some live execution $\alpha \in \mathit{lexecs}(A, L)$. By the Live Execution Correspondence Theorem (2), there exists a live execution $\alpha' \in \mathit{lexecs}(B, M)$ such that $(\alpha, \alpha') \in S^\ell$. Since $(\alpha, \alpha') \in S^\ell$, we have $(\alpha, \alpha') \in g$ by Definitions 2 and 12. Hence, by Lemma 1, $\mathit{trace}(\alpha) = \mathit{trace}(\alpha')$. Hence $\beta = \mathit{trace}(\alpha')$, and so $\beta \in \mathit{traces}(\mathit{lexecs}(B, M))$, since $\alpha' \in \mathit{lexecs}(B, M)$. Since β was chosen arbitrarily, we conclude $\mathit{traces}(\mathit{lexecs}(A, L)) \subseteq \mathit{traces}(\mathit{lexecs}(B, M))$, as desired.

5 Refining Liveness Properties Within the Same Level of Abstraction

The previous section showed how to refine an abstract liveness condition M to a concrete liveness condition L: every pair $q \in M$ is mapped into some pair p in the semantic closure $\widehat{L}$ of L, and then a liveness-preserving simulation relation that relates the R and G sets of p, q appropriately is devised. We assume that the liveness properties L, M are directly specified, and so the pairs in M and in L are easy to identify.[5] However, pairs in $\widehat{L} - L$ are not directly specified, but only given implicitly by A, L, and Definition 8. Thus, the question arises, given a pair $q \in M$ that is mapped to some pair p, how do we establish $p \in \widehat{L}$? We do so as follows.

Given such a pair p, we refine it into a finite "lattice" of pairs that are already known to be in $\widehat{L}$. Let P be a finite subset of $\widehat{L}$, and let $\prec$ be an irreflexive partial order over P[6]. If $r \in P$, define $\mathit{succ}(r) = \{w \in P \mid r \prec w \wedge \forall w' : r \preceq w' \prec w \Rightarrow r = w'\}$, where $r \preceq w \overset{\mathrm{df}}{=} r \prec w$ or $r = w$. Thus, $\mathit{succ}(r)$ is the set of all "immediate successors" of r in $(P, \prec)$. We now impose two technical conditions on P: (1) for every pair r, the G set of r must be a subset of the union of the R sets of all the immediate successors of r, i.e., $r.\mathsf{G} \subseteq \bigcup_{w \in \mathit{succ}(r)} w.\mathsf{R}$, and (2) P has a single $\prec$-minimum element $\mathit{bottom}(P)$, and a single $\prec$-maximum element $\mathit{top}(P)$, and $\mathit{bottom}(P).\mathsf{R} = p.\mathsf{R}$ and $\mathit{top}(P).\mathsf{G} = p.\mathsf{G}$.

Now let α be an arbitrary live execution of (A, L). Then, $\alpha \models \Box\Diamond r.\mathsf{R} \Rightarrow \Box\Diamond r.\mathsf{G}$ and $\alpha \models \Box\Diamond w.\mathsf{R} \Rightarrow \Box\Diamond w.\mathsf{G}$, for all $w \in \mathit{succ}(r)$. Since $\mathit{succ}(r)$ is finite and $r.\mathsf{G} \subseteq \bigcup_{w \in \mathit{succ}(r)} w.\mathsf{R}$, it follows that, if $r.\mathsf{G}$ holds infinitely often in α, then $w.\mathsf{R}$ holds infinitely often in α, for some $w \in \mathit{succ}(r)$. Hence, by "chaining" the above implications, we get $\alpha \models \Box\Diamond r.\mathsf{R} \Rightarrow \Box\Diamond \bigcup_{w \in \mathit{succ}(r)} w.\mathsf{G}$. Thus, $\langle\!\langle r.\mathsf{R}, \bigcup_{w \in \mathit{succ}(r)} w.\mathsf{G} \rangle\!\rangle \in \widehat{L}$ by Definition 8. Thus, the $\prec$ ordering provides a way of relating the complemented-pairs of P so that the complemented-pairs property (infinitely often R implies infinitely often G) can be generalized to encompass a pair and its immediate successor pairs. By starting with the $\prec$-minimum pair $\mathit{bottom}(P)$, and applying the above argument inductively

[5] For example, if we were attempting to mechanize our method, we would assume that M, L are recursive sets.

[6] Following convention, we shall refer to this ordered set simply as P when no confusion arises.

(using $\prec$ as the underlying ordering), we can establish the complemented-pairs property for $\langle bottom(P).\mathsf{R}, top(P).\mathsf{G}\rangle$, i.e., $\alpha \models \Box\Diamond bottom(P).\mathsf{R} \Rightarrow \Box\Diamond top(P).\mathsf{G}$, and so $\langle bottom(P).\mathsf{R}, top(P).\mathsf{G}\rangle \in \widehat{L}$. Since we require $bottom(P).\mathsf{R} = p.\mathsf{R}$ and $top(P).\mathsf{G} = p.\mathsf{G}$, we obtain the desired result that $p \in \widehat{L}$.

Definition 14 (Complemented-pairs Lattice) Let (A, L) be a live automaton. Then $(P, \prec)$ is a *complemented-pairs lattice over* $\widehat{L}$ iff[7]

1. P is a finite subset of $\widehat{L}$,
2. $\prec$ is an irreflexive partial order over P,
3. P contains an element $top(P)$ which satisfies $\forall r \in P : r \preceq top(P)$, and an element $bottom(P)$ which satisfies $\forall r \in P : bottom(P) \preceq r$, and
4. $\forall r \in P - \{top(P)\} : r.\mathsf{G} \subseteq \bigcup_{w \in succ(r)} w.\mathsf{R}$.

The elements $top(P)$ and $bottom(P)$ are necessarily unique, since $\prec$ is a partial order. Let $lattices(\widehat{L})$ denote the set of all complemented-pairs lattices over $\widehat{L}$.

Lemma 6 *Let (A, L) be a live automaton, $(P, \prec) \in lattices(\widehat{L})$, $\bot = bottom(P)$, and $\top = top(P)$. Then $\langle \bot.\mathsf{R}, \top.\mathsf{G}\rangle \in \widehat{L}$.*

Proof Let α be an arbitrary live execution of (A, L). We show $\alpha \models \Box\Diamond\bot.\mathsf{R} \Rightarrow \Box\Diamond\top.\mathsf{G}$. By Definition 8, this establishes the lemma.

We assume $\alpha \models \Box\Diamond\bot.\mathsf{R}$ and establish $\alpha \models \Box\Diamond\top.\mathsf{G}$. First, we establish:

If $r \in P$, $r \neq \top$, and $\alpha \models \Box\Diamond r.\mathsf{R}$, then $\alpha \models \Box\Diamond w.\mathsf{R}$ for some $w \in succ(r)$. (*)

Proof of (*): Assume the antecedent of (*). Since α is live and $r \in \widehat{L}$, we have $\alpha \models \Box\Diamond r.\mathsf{R} \Rightarrow \Box\Diamond r.\mathsf{G}$ by Definition 8. Hence $\alpha \models \Box\Diamond r.\mathsf{G}$. By Definition 14, $r.\mathsf{G} \subseteq \bigcup_{w \in succ(r)} w.\mathsf{R}$. Hence $\alpha \models \Box\Diamond \bigcup_{w \in succ(r)} w.\mathsf{R}$. Since P is finite, $succ(r)$ is finite. It follows that $\alpha \models \Box\Diamond w.\mathsf{R}$ for some $w \in succ(r)$. (End of proof of (*).)

We now construct a sequence $r_1, r_2, \ldots, r_i, \ldots$ of pairs in P such that $\forall i \geq 1 : \alpha \models \Box\Diamond r_i.\mathsf{R}$. We let $r_1 = \bot$, noting that $\alpha \models \Box\Diamond\bot.\mathsf{R}$ by assumption. We derive r_{i+1} by applying (*) to r_i. It follows by induction on the length of the derived sequence that $\alpha \models \Box\Diamond r_{i+1}.\mathsf{R}$ ($r_1 = \bot$ supplies the base case). Now suppose $\top$ is not in $r_1, r_2, \ldots$ Then (*) can be applied indefinitely. Since $r_{i+1} \in succ(r_i)$, it follows that $r_j \prec r_{i+1}$ for all $j \in 1..i$. Hence $r_j \neq r_{i+1}$ for all $j \in 1..i$. Thus $r_1, r_2, \ldots$ is an infinite sequence of pairwise different complemented-pairs in P. But this is impossible, since P is finite. Hence the assumption that $\top$ is not in $r_1, r_2, \ldots$ is false. It follows that $r_1, r_2, \ldots$ is a finite sequence of pairwise different complemented-pairs, with $\top$ as its last member. Hence $\alpha \models \Box\Diamond\top.\mathsf{R}$. Since α is live and $\top \in \widehat{L}$, $\alpha \models \Box\Diamond\top.\mathsf{R} \Rightarrow \Box\Diamond\top.\mathsf{G}$. Hence $\alpha \models \Box\Diamond\top.\mathsf{G}$, as desired.

We remark that when constructing a lattice to refine a complemented-pair, we can use requirement 4 of Definition 14 ($r.\mathsf{G} \subseteq \bigcup_{w \in succ(r)} w.\mathsf{R}$) as a constraint that suggests how to order the complemented-pairs of the lattice. Also, while Lemma 6 presents one method of establishing the membership of complemented-pairs in $\widehat{L}$, our overall methodology is not restricted to this particular method. Any appropriate deductive technique that suffices can be used, for example that of [41], which is based on linear temporal logic. This provides a way of using deductive methods generally, and those based on temporal logic in particular, within a framework which accommodates the refinement of liveness properties across multiple levels of abstraction.

[7] Note that we use the term "lattice" in an informal sense, since our complemented-pairs lattices do not satisfy the mathematical definition of a lattice.

6 Example—The Eventually Serializable Data Service

The eventually-serializable data service (ESDS) of [15,27] is a replicated, distributed data service that trades off immediate consistency for improved efficiency. A shared data object is replicated, and the response to an operation at a particular replica may be out of date, i.e., not reflecting the effects of other operations that have not yet been received by that replica. Thus, operations may be reordered *after* the response is issued. Replicas communicate amongst each other the operations they receive, so that eventually every operation "stabilizes," i.e., its ordering is fixed with respect to all other operations. Clients may require an operation to be *strict*, i.e., stable at the time of response, and so it cannot be reordered after the response is issued. Clients may also specify, in an operation x, a set $x.prev$ of other operations that should precede x (client-specified constraints, CSC). We let $\mathcal{O}$ be the (countable) set of all operations on the data object, and V be the set of all possible results of operations in $\mathcal{O}$. $\mathcal{R}$ is the set of all replicas, and $client(x)$ is the client issuing operation x. We use x, y to index over operations, c to index over clients, and r, r', i to index over replicas. Each operation x has a unique identifier $x.id$. $\mathcal{I}$ is the set of identifiers of operations in $\mathcal{O}$.

In Appendix C, we give the I/O automata code (in "precondition-effect" style) from [15]. I/O automata [34] add an input/output distinction to the external actions, i.e, all external actions of an automaton are either input actions (which must furthermore be enabled in all states), or output actions. This is needed to define a parallel composition operator $\|$ with good compositional properties. Figure 7 gives the environment of the ESDS system: a set of users, or clients, which output requests $\mathsf{request}(x)$ to perform operations x, and input responses $\mathsf{response}(x, v)$ to the requests, with returned value v. Figure 8 presents the specification *ESDS-I*. As a high-level specification, *ESDS-I* is a single automaton, and therefore it does not address issues of concurrency and distribution. The only concern is to specify the set of correct traces, which are by definition the traces of *ESDS-I*. *ESDS-I* inputs requests $\mathsf{request}(x)$, and outputs responses $\mathsf{response}(x, v)$ to the requests, with returned value v. Once $\mathsf{request}(x)$ has been received, it is "entered" into the current partial order po, via internal action $\mathsf{enter}(x, \textit{new-po})$, which updates the value of po to that given by *new-po*. This new value must include all operations in $x.prev$, and all operations that have stabilized, as preceding x. Note that $span(R) = \{x \mid xRy \vee yRx\}$, where R is a binary relation. At any time, it is permissible to impose new ordering constraints, which is done by internal action $\mathsf{add\_constraints}(\textit{new-po})$. The $\mathsf{stabilize}(x)$ internal action checks that x is totally ordered with respect to all other operations ($\forall y \in ops,\ y \preceq_{po} x \vee x \preceq_{po} y$), and that all operations that precede x have already stabilized ($ops|_{\prec_{po} x} \subseteq stabilized$). In this case, x itself can be stabilized. The $\mathsf{calculate}(x, v)$ internal action computes a return value v for the operation x. If x is strict, then $\mathsf{calculate}(x, v)$ checks (in its precondition) that x has stabilized. The $valset(x, ops, \prec_{po})$ function returns the set of all values for x which are consistent with the set ops of all operations that have been entered, and the partial order $\prec_{po}$ defined by po. The actual value returned is then chosen nondeterministically from this set.

As an intermediate step, we refine *ESDS-I* to a second level specification *ESDS-II*. This refinement consists only of changing some of the transitions. The state space and the signature remain the same. Figure 9 presents these changes, as changes to the "precondition-effect" definitions of some of the actions in the action list. The main difference with *ESDS-I* is that the precondition to stabilize an operation x is relaxed: now, all operations that precede x are not required to be stable themselves, but are only required to be totally ordered with respect to all other entered operations ($\prec_{po}$ totally orders $ops|_{\prec_{po}x}$). This intermediate version *ESDS-II* is useful, as it is easier to construct a simulation from the implementation to *ESDS-II*, and another simulation from *ESDS-II* to *ESDS-I*, than it is to construct a simulation from the implementation directly to *ESDS-I*.

The implementation consists of front-ends, replicas, and channels. Each client c has a front-end $\mathit{Frontend}(c)$, see Figure 10, which inputs requests $\mathsf{request}(x)$, and relays them onto one or more of the replicas $\mathit{Replica}(r)$, via output action $\mathsf{send}_{cr}(\langle\text{“request”}, x\rangle)$. $\mathit{Frontend}(c)$ receives responses from the replicas via input action $\mathsf{receive}_{rc}(\langle\text{“response”}, x, v\rangle)$, and relays the response onto the client via output action $\mathsf{response}(x, v)$. While the frontend can receive several replies for x from various replicas, it only relays one of these onto the client. A replica r (Figure 11) receives requests to perform operation x via input action $\mathsf{receive}_{cr}(\langle\text{“request”}, x\rangle)$. It queues received operations into a set $\mathit{pending}_r$ of pending operations. A pending operation x can be “performed” by the internal action $\mathsf{do\_it}_r(x, l)$ if all operations in $x.\mathit{prev}$ have been performed. In this case, x is assigned a “label” l larger than the labels of all operations known to be done at replica r. This label determines the values that can be returned for x, using the *valset* function. Once x has been processed by $\mathsf{do\_it}_r(x, l)$, a value v for x can be returned by the output action $\mathsf{send}_{rc}(\langle\text{“response”}, x, v\rangle)$. v is nondeterministically chosen from among the set returned by $\mathit{valset}(x, \mathit{done}_r[r], \prec_{lc_r})$, which computes all values for x that are consistent with the set $\mathit{done}_r[r]$ of operations done at replica r, and the partial order $\prec_{lc_r}$ on operations that is determined by the labels assigned to each operation. In addition, replicas “gossip” amongst each other, by means of the actions $\mathsf{send}_{rr'}(\langle\text{“gossip”}, R, D, L, S\rangle)$ and $\mathsf{receive}_{r'r}(\langle\text{“gossip”}, R, D, L, S\rangle)$. The purpose of gossiping is to bring each other up to date on the operations that they have executed. All communication between the front-ends and the replicas is by means of reliable asynchronous channels. Figure 12 shows a channel from process i to process j with messages drawn from some set $\mathcal{M}$.

We will use *ESDS-Alg* to refer to the parallel composition of all replicas, front-ends, and channels, with all send and $\mathsf{receive}$ actions hidden[8]. Since the users must be taken into account, the first-level specification, second-level specification, and implementation are the I/O automata $\mathit{ESDS\text{-}I} \parallel \mathit{Users}$, $\mathit{ESDS\text{-}II} \parallel \mathit{Users}$, and $\mathit{ESDS\text{-}Alg} \parallel \mathit{Users}$, respectively. We refer the reader to [15] for a complete description of the ESDS system.

The liveness condition used in (the conference version of) [15] is that every request should eventually receive a response, and every operation should stabilize. We express this as the following complemented-pairs liveness condition *M-I* for the specification $\mathit{ESDS\text{-}I} \parallel \mathit{Users}$:[9]

- $\{\langle\!\langle x \in \mathit{wait}, x \notin \mathit{wait}\rangle\!\rangle \mid x \in \mathcal{O}\}$, i.e., every request eventually receives a response.
- $\{\langle\!\langle x \in \mathit{wait}, x \in \mathit{stabilized}\rangle\!\rangle \mid x \in \mathcal{O}\}$, i.e., every operation eventually stabilizes.

Because the number of submitted operations x in general grows without bound with time, a countably infinite number of pairs is needed to express this liveness condition in the natural manner illustrated above. Note that we use predicates to denote sets of states.

[8] I/O automata composed in parallel synchronize on actions with the same name, and otherwise execute independently. An action is hidden by removing it from the set of output actions and adding it to the set of internal actions. We refer the reader to [15, section 3] for formal definitions of parallel composition and hiding.

[9] Throughout this section, our notation is consistent with [15].

G is a relation between states in *ESDS-II* $\|$ *Users* and *ESDS-I* $\|$ *Users*, such that $(s, u) \in G$ if and only if $s \in \mathit{states}(\mathit{ESDS\text{-}II} \,\|\, \mathit{Users})$, $u \in \mathit{states}(\mathit{ESDS\text{-}I} \,\|\, \mathit{Users})$, and:

- $u.\mathit{wait} = s.\mathit{wait}$
- $u.\mathit{rept} = s.\mathit{rept}$
- $u.\mathit{ops} = s.\mathit{ops}$
- $u.\mathit{po} = s.\mathit{po}$
- $u.\mathit{stabilized} \supseteq s.\mathit{stabilized}$

Fig. 3 Forward Simulation from *ESDS-II* $\|$ *Users* to *ESDS-I* $\|$ *Users*

6.1 Refinement from *ESDS-I* $\|$ *Users* to *ESDS-II* $\|$ *Users*

The top-level specification *ESDS-I* $\|$ *Users* and second-level specification *ESDS-II* $\|$ *Users* have the same state-space, they only differ in some actions, as shown in Figure 9. Hence, we let the liveness condition *M-II* of *ESDS-II* $\|$ *Users* consist of the same complemented-pairs as those in *M-I*, and we map each pair of *M-I* into the same pair of *M-II*.

In [15], it is shown that the relation G given in Figure 3 is a forward simulation relation from *ESDS-II* $\|$ *Users* to *ESDS-I* $\|$ *Users*. We show that G is also a liveness-preserving forward simulation. For the pair $\langle\!\langle x \in \mathit{wait}, x \notin \mathit{wait} \rangle\!\rangle$ it is clear that G satisfies clause 2 of Definition 10, since G only relates states that agree on the value of *wait*. For the pair $\langle\!\langle x \in \mathit{wait}, x \in \mathit{stabilized} \rangle\!\rangle$, we see from Figure 3 that if $s \in \mathit{states}(\mathit{ESDS\text{-}II} \,\|\, \mathit{Users})$ and $u \in \mathit{states}(\mathit{ESDS\text{-}I} \,\|\, \mathit{Users})$ are related by G, and s satisfies $x \in \mathit{stabilized}$, then u also satisfies $x \in \mathit{stabilized}$, since $s.\mathit{stabilized} \subseteq u.\mathit{stabilized}$. Since s and u agree on the value of *wait*, we conclude that G satisfies clause 2 of Definition 10, for this pair too.

By inspection, we verify that, in every live execution of *ESDS-II*, there is an infinite number of executions of non-stabilize actions. Now according to the definition of G in Figure 3, every action in *ESDS-II* is simulated by the same action in *ESDS-I*, except for the stabilize action; a single stabilize(x) action in *ESDS-II* can be simulated by a possibly empty sequence of stabilize actions in *ESDS-I*. Hence, any transition generated by executing any action other than stabilize is not always-silent, by clause 3 of Definition 10. Since every live execution of *ESDS-II* contains an infinite number of these transitions, clause 3 of Definition 10 is satisfied.

Since each pair of *M-I* is mapped into a pair of *M-II* itself, rather than the semantic closure $\widehat{\mathit{M\text{-}II}}$ of *M-II*, we are done (i.e., there is no need to construct complemented-pairs lattices for these pairs).

Since Definition 10 is now satisfied, we have established $(\mathit{ESDS\text{-}II} \,\|\, \mathit{Users}, \mathit{M\text{-}II}) \leq_{\ell F} (\mathit{ESDS\text{-}I} \,\|\, \mathit{Users}, \mathit{M\text{-}I})$. Hence, applying Theorem 3, we conclude $(\mathit{ESDS\text{-}II} \,\|\, \mathit{Users}, \mathit{M\text{-}II}) \sqsubseteq_{\ell} (\mathit{ESDS\text{-}I} \,\|\, \mathit{Users}, \mathit{M\text{-}I})$.

6.2 Refinement from *ESDS-II* $\|$ *Users* to *ESDS-Alg* $\|$ *Users*

Let L be the liveness condition of *ESDS-Alg* $\|$ *Users*. Since *ESDS-Alg* $\|$ *Users* is an implementation, we take L to be the following: every action that is continuously enabled from some point onwards is eventually executed (fair scheduling), and every message that is sent is eventually received (fair polling of channels). These are reasonable liveness properties to expect of an implementation.

F is a relation between states in *ESDS-Alg* $\|$ *Users* and *ESDS-II* $\|$ *Users*, i.e., $F \subseteq$ *states*(*ESDS-Alg* $\|$ *Users*) $\times$ *states*(*ESDS-II* $\|$ *Users*), such that $(s,u) \in F$ if and only if:

- $u.requested = s.requested$
- $u.responded = s.responded$
- $u.wait = \bigcup_c s.wait_c$
- $u.rept = \bigcup_c s.rept_c \cup s.potential\_rept_c$
- $u.ops = s.ops = \bigcup_r s.done_r[r]$
- $u.po \subseteq s.po$
- $u.stabilized = \bigcap_r s.stable_r[r]$

where $s.potential\_rept_c = \{(x,v) \mid \langle\text{“response”}, x, v\rangle \in \bigcup_r s.channel_{rc} \wedge s.wait_c\}$ is the set of responses en route to *Frontend*(c), and $u.po$ is the partial order induced by the various operation constraints in the implementation. See [15] for details.

Fig. 4 Forward simulation from *ESDS-Alg* $\|$ *Users* to *ESDS-II* $\|$ *Users*

We map the pair $\langle\!\langle x \in wait, x \notin wait\rangle\!\rangle$ of *M-II* into the pair $\langle\!\langle x \in wait_c, x \notin wait_c\rangle\!\rangle$, where $c = client(x)$ is the client that requests operation x. We map the pair $\langle\!\langle x \in wait, x \in stabilized\rangle\!\rangle$ of *M-II* into the pair $\langle\!\langle x \in wait_c, x \in \bigcap_i stable_i[i]\rangle\!\rangle$.

The proof obligations are then to exhibit a liveness-preserving forward simulation for this choice of pair-mapping, and to show that the pairs $\langle\!\langle x \in wait_c, x \notin wait_c\rangle\!\rangle$ and $\langle\!\langle x \in wait_c, x \in \bigcap_i stable_i[i]\rangle\!\rangle$ are members of $\widehat{L}$, since they are not members of L.

6.2.1 Establishing a Liveness-preserving Forward Simulation

In [15], it is shown that the relation F given in Figure 4 is a forward simulation relation from *ESDS-Alg* $\|$ *Users* to *ESDS-II* $\|$ *Users*. We establish that F is also a liveness-preserving forward simulation. We first

By Definition 19, F already satisfies clause 1 of Definition 10. We argue that F also satisfies clauses 2 and 3. Let $SpReq = \langle\!\langle x \in wait, x \notin wait\rangle\!\rangle$, $ImpReq = \langle\!\langle x \in wait_c, x \notin wait_c\rangle\!\rangle$, $SpStab = \langle\!\langle x \in wait, x \in stabilized\rangle\!\rangle$, $ImpStab = \langle\!\langle x \in wait_c, x \in \bigcap_i stable_i[i]\rangle\!\rangle$. Let $B =$ *ESDS-II* $\|$ *Users*, and $A =$ *ESDS-Alg* $\|$ *Users*. Let s, u range over the states of *ESDS-Alg* $\|$ *Users*, *ESDS-II* $\|$ *Users* respectively. We use the notation $s.v$ to denote the value of state variable v in state s, and likewise for $u.v$.

Establishing clause 2 of Definition 10 for the pairs $SpReq = \langle\!\langle x \in wait, x \notin wait\rangle\!\rangle \in M\text{-}I$ *and* $ImpReq = \langle\!\langle x \in wait_c, x \notin wait_c\rangle\!\rangle$. F relates states s and u only if $u.wait = \bigcup_c s.wait_c$. Hence $x \in u.wait$ iff $x \in s.wait_c$, where $c = client(x)$. Thus u is a $SpReq.\mathsf{R}$ state iff s is a $ImpReq.\mathsf{R}$ state, and u is a $SpReq.\mathsf{G}$ state iff s is a $ImpReq.\mathsf{G}$ state.

Let $s \xrightarrow{a}_A s'$ and consider all possibilities for a. If a is one of send (along any channel), $\mathsf{receive}$ (from any channel), or $\mathsf{do\_it}_r$ (for any replica r), then a does not change $wait_c$ (for any client c), and the actions of *ESDS-II* $\|$ *Users* that simulate a do not change $wait$. Hence if $u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B u_2 \xrightarrow{b_3}_B \cdots \xrightarrow{b_n}_B u_n$ is the simulating execution fragment of *ESDS-II* $\|$ *Users*, corresponding to $s \xrightarrow{a}_A s'$ for the aforementioned cases of a, then we immediately conclude that (1) all $u_i, i \in 0 \ldots n$ have the same value of $wait$, and (2) s and s' have the same value of $\bigcup_c wait_c$. Together with $u_0.wait = \bigcup_c s.wait_c$, this allows us to conclude $(\exists i \in 0 \ldots n : u_i \in SpReq.\mathsf{R})$ iff $s \in ImpReq.\mathsf{R}$ or $s' \in ImpReq.\mathsf{R}$, and $s \in ImpReq.\mathsf{G}$ or $s' \in ImpReq.\mathsf{G}$ iff $(\exists i \in 0 \ldots n : u_i \in SpReq.\mathsf{G})$. Thus clause 2 of Definition 10 is satisfied in this case.

If a is $\mathsf{request}(x)$, this is simulated by the same action in *ESDS-II* $\|$ *Users*. $\mathsf{request}(x)$ adds x to $wait_c$ in *ESDS-Alg* $\|$ *Users*, and adds x to $wait$ in *ESDS-II* $\|$ *Users*. Hence, using similar reasoning as above, we easily verify that clause 2 of Definition 10 is satisfied in this case. The argument for $a = \mathsf{response}(x, v)$ is similar. This concludes our argument that clause 2 of Definition 10 holds for the pairs *SpReq* and *ImpReq*.

Establishing clause 2 of Definition 10 for the pairs $SpStab = \langle x \in wait, x \in stabilized\rangle \in$ *M-I and* $ImpStab = \langle x \in wait_c, x \in \bigcap_i stable_i[i]\rangle$. F relates states s and u only if $u.wait = \bigcup_c s.wait_c$ and $u.stabilized = \bigcap_i s.stable_i[i]$ (definition of F in [15], and Figure 4). Hence $x \in u.wait$ iff $x \in s.wait_c$, where $c = client(x)$, and $x \in u.stabilized$ iff $x \in \bigcap_i s.stable_i[i]$. Thus $u \in SpStab.\mathsf{R}$ iff $s \in ImpStab.\mathsf{R}$, and $u \in SpStab.\mathsf{G}$ iff $s \in ImpStab.\mathsf{G}$.

Let $s \xrightarrow{a}_A s'$ and let $u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B u_2 \xrightarrow{b_3}_B \cdots \xrightarrow{b_n}_B u_n$ be the execution fragment of *ESDS-II* $\|$ *Users* that simulates $s \xrightarrow{a}_A s'$. Given the previous remarks, we conclude immediately that clause 2 of Definition 10 is satisfied when $u_1, \ldots, u_{n-1}$ are not present, i.e., the simulating fragment consists of either a single state or a single transition.

The only case where $u_0 \xrightarrow{b_1}_B u_1 \xrightarrow{b_2}_B u_2 \xrightarrow{b_3}_B \cdots \xrightarrow{b_n}_B u_n$ consists of more than one transition is when $a = \mathsf{receive}_{rr'}(m)$. In this case, the actions $b_1, \ldots, b_n$ are $\mathsf{add\_constraints}(s'.po), \mathsf{stabilize}(x_1), \ldots, \mathsf{stabilize}(x_k)$, where $\{x_1, \ldots, x_k\} = \bigcap_i s'.stable_i[i]$ (see [15], Section 8). Now $\bigcap_i s.stable_i[i] \subseteq \bigcap_i s'.stable_i[i]$ by inspection of the $\mathsf{receive}_{rr'}(m)$ action in Figure 11. Also, $u_0.stabilized = \bigcap_i s.stable_i[i]$, and $u_n.stabilized = \bigcap_i s'.stable_i[i] = \{x_1, \ldots, x_k\}$, by definition of F and $x_1, \ldots, x_k$.

Now $\mathsf{receive}_{rr'}(m)$ does not affect $wait_c$, and $\mathsf{add\_constraints}(s'.po), \mathsf{stabilize}(x_1), \ldots,$ $\mathsf{stabilize}(x_k)$ do not affect $wait$. Hence, $(\exists i \in 0 \ldots n : u_i \in SpStab.\mathsf{R})$ iff $s \in ImpStab.\mathsf{R}$ or $s' \in ImpStab.\mathsf{R}$. Also, suppose $s \in ImpStab.\mathsf{G}$ or $s' \in ImpStab.\mathsf{G}$, i.e., $x \in \bigcap_i s.stable_i[i]$ or $x \in \bigcap_i s'.stable_i[i]$. Hence $x \in \bigcap_i s'.stable_i[i]$ since $\bigcap_i s.stable_i[i] \subseteq \bigcap_i s'.stable_i[i]$. Since $u_n.stabilized = \bigcap_i s'.stable_i[i]$, we have $x \in u_n.stabilized$. Hence $u_n \in SpStab.\mathsf{G}$. Hence $(\exists i \in 0 \ldots n : u_i \in SpStab.\mathsf{G})$.

We have thus established clause 2 of Definition 10 for the pairs *SpStab* and *ImpStab*.

Establishing clause 3 of Definition 10. From Figure 11, it is clear that the action $\mathsf{send}_{rr'}(m)$ (for some m) is continuously enabled, and hence executed infinitely often in any live execution of *ESDS-Alg* $\|$ *Users*. Hence, the action $\mathsf{receive}_{rr'}(m)$ is also executed infinitely often. Now, according to the definition of F (see [15], Section 8), $\mathsf{receive}_{rr'}(m)$ is simulated by the sequence of actions $\mathsf{add\_constraints}(s'.po)$, $\mathsf{stabilize}(x_1)$, ..., $\mathsf{stabilize}(x_k)$, where $\{x_1, \ldots, x_k\} = \bigcap_i s'.stable_i[i]$, and s' is the state of *ESDS-Alg* $\|$ *Users* resulting from the execution of $\mathsf{receive}_{rr'}(m)$. Thus, $\mathsf{receive}_{rr'}(m)$ is always matched by at least one action, namely $\mathsf{add\_constraints}(s'.po)$. Hence, any transition generated by executing $\mathsf{receive}_{rr'}(m)$ is not always-silent, by clause 3 of Definition 10. Since every live execution of *ESDS-Alg* $\|$ *Users* contains an infinite number of these transitions, clause 3 of Definition 10 is satisfied.

6.2.2 Establishing Membership in $\widehat{L}$

Establishing $\langle x \in wait_c, x \notin wait_c\rangle \in \widehat{L}$. We use a complemented-pairs lattice over $\widehat{L}$, together with Lemma 6, to establish $\langle x \in wait_c, x \notin wait_c\rangle \in \widehat{L}$. Recall that L is the complemented-pairs liveness condition for the implementation *ESDS-Alg* $\|$ *Users*.

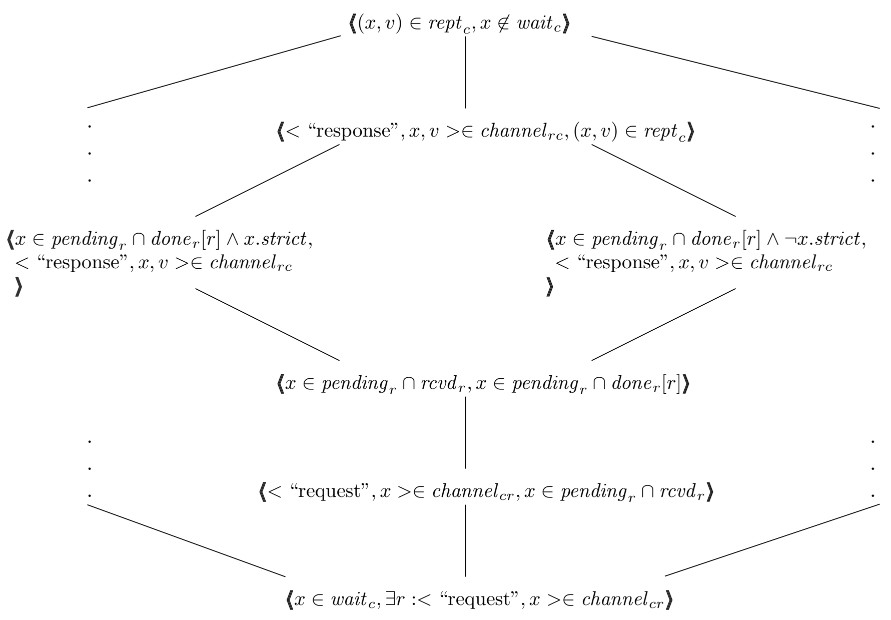
$\langle (x,v) \in rept_c, x \notin wait_c \rangle$
$\langle <\text{“response”}, x, v> \in channel_{rc}, (x,v) \in rept_c \rangle$
$\langle x \in pending_r \cap done_r[r] \wedge x.strict, <\text{“response”}, x, v> \in channel_{rc} \rangle$
$\langle x \in pending_r \cap done_r[r] \wedge \neg x.strict, <\text{“response”}, x, v> \in channel_{rc} \rangle$
$\langle x \in pending_r \cap rcvd_r, x \in pending_r \cap done_r[r] \rangle$
$\langle <\text{“request”}, x> \in channel_{cr}, x \in pending_r \cap rcvd_r \rangle$
$\langle x \in wait_c, \exists r :< \text{“request”}, x> \in channel_{cr} \rangle$

Fig. 5 Complemented-pairs lattice that establishes $\langle x \in wait_c, x \notin wait_c \rangle \in \widehat{L}$ ($c = client(x)$).

At the implementation level, the natural liveness hypothesis is that each continuously enabled action is eventually executed, and each message in transit eventually arrives. We use this hypothesis to justify the pairs in L (which are also in $\widehat{L}$, by definition). Figure 5 shows the complemented-pairs lattice that we use. $c = client(x)$ is the client that invoked operation x. We display the portion of the lattice corresponding to a single replica r. The $\vdots$ indicate where isomorphic copies corresponding to the other replicas occur (the number of replicas is finite). Let L consist of all the pairs in Figure 5. It is straightforward to verify that Figure 5 satisfies all the conditions of Definition 14. We justify the complemented-pairs in Figure 5 as follows:

1. $\langle x \in wait_c, \exists r :< \text{“request”}, x> \in channel_{cr} \rangle$.
   $send_{cr}$ is continuously enabled and eventually happens, for at least one replica r.
2. $\langle <\text{“request”}, x> \in channel_{cr}, x \in pending_r \cap rcvd_r \rangle$.
   Liveness of $channel_{cr}$, and the definition of action $\mathsf{receive}_{cr}$ in Figure 11.
3. $\langle x \in pending_r \cap rcvd_r, x \in pending_r \cap done_r[r] \rangle$.
   If $x.prev \subseteq done_r[r]$ holds continuously, then either $\mathsf{do\_it}_r$ is continuously enabled and eventually happens (making $x \in done_r[r]$ true), or $\mathsf{do\_it}_r$ is disabled because $x \in done_r[r]$ becomes true due to a gossip message. Establishing $x.prev \subseteq done_r[r]$ essentially requires a “sublattice” for each $x' \in x.prev$. This sublattice is a “chain” consisting of three pairs, with the ordering (a) $\prec$ (b) $\prec$ (c):
   (a) $\langle x \in pending_r \cap rcvd_r, x' \in pending_{r'} \cap rcvd_{r'} \rangle$ is the bottom element. It is justified since each client includes in $x.prev$ only operations that have already been requested. Thus $x' \in x.prev$ is eventually received by some replica r', at which point $x' \in pending_{r'} \cap rcvd_{r'}$ holds.

(b) $\langle x' \in pending_{r'} \cap rcvd_{r'}, x' \in pending_{r'} \cap done_{r'}[r'] \rangle$ is the middle element. It is justified "inductively," i.e., it can be expanded into a sublattice in exactly the same way as $\langle x \in pending_r \cap rcvd_r, x \in pending_r \cap done_r[r] \rangle$. This "nested" expansion is guaranteed to terminate however, since $x.prev$ is finite, for all x.

(c) $\langle x' \in done_{r'}[r'], x' \in done_r[r] \rangle$ is the top element. It is justified since r' eventually sends a gossip message to r.

By applying Lemma 6 to this sublattice, we conclude $\langle x \in pending_r \cap rcvd_r, x' \in done_r[r] \rangle \in \widehat{L}$. Now $done_r[r]$ increases monotonically, $x' \in done_r[r]$ is stable—once true, it remains true. Hence, from the aforementioned pair for each $x' \in x.prev$, we conclude that $x.prev \subseteq done_r[r]$ eventually holds, and remains true subsequently, as required.

Note that the condition $l > label_r(y.id)$ does not need to be verified as eventually holding, since it merely expresses a constraint on the value of the "action parameter" l, i.e., the only instances of $\mathsf{do\_it}_r(x, l)$ which are enabled are those having values of l that satisfy $l > label_r(y.id)$. That is, l is properly regarded as part of the "name" of the action $\mathsf{do\_it}_r(x, l)$.

4. $\langle x \in pending_r \cap done_r[r] \wedge x.strict, <\text{"response"}, x, v> \in channel_{rc} \rangle$.
 This is justified by the following sublattice, where the ordering relation is (a) $\prec$ (b) $\prec$ (c) $\prec$ (d) $\prec$ (e). $x \in pending_r$, $x.strict$, are implicit conjuncts of all the predicates in the sublattice, except the Green predicate of pair (e), and are omitted for clarity.
 (a) $\langle x \in \cap done_r[r], x \in \cap_{r'} done_{r'}[r'] \rangle$. Justified since r sends gossip messages to every other replica r'.
 (b) $\langle x \in \cap_{r'} done_{r'}[r'], x \in stable_r[r] \rangle$. Justified since each r' sends gossip messages to r.
 (c) $\langle x \in stable_r[r], x \in \cap_{r'} stable_{r'}[r'] \rangle$. Justified since r sends gossip messages to every other replica r'.
 (d) $\langle x \in \cap_{r'} stable_{r'}[r'], x \in \cap_{r'} stable_r[r'] \rangle$. Justified since each r' sends gossip messages to r.
 (e) $\langle x \in \cap_{r'} stable_r[r'], <\text{"response"}, x, v> \in channel_{rc} \rangle$. Justified since $x \in pending_r$, $x \in done_r[r]$, and $x \in \cap_{r'} stable_r[r']$ all hold continuously, since $done_r[r]$ and $stable_r[r']$ grow monotonically. Hence $send_{rc}(<\text{"response"}, x, v>)$ is continuously enabled, and so is eventually executed.
5. $\langle x \in pending_r \cap done_r[r] \wedge \neg x.strict, <\text{"response"}, x, v> \in channel_{rc} \rangle$.
 $send_{rc}(<\text{"response"}, x, v>)$ is continuously enabled and eventually happens.
6. $\langle <\text{"response"}, x, v> \in channel_{rc}, (x, v) \in rept_c \rangle$.
 Liveness of $channel_{rc}$, and the definition of action $\mathsf{receive}_{rc}$ in Figure 10.
7. $\langle (x, v) \in rept_c, x \notin wait_c \rangle$.
 $\mathsf{response}(x, v)$ is continuously enabled and eventually happens.

Establishing $\langle x \in wait_c, x \in \bigcap_i stable_i[i] \rangle \in \widehat{L}$. We use the complemented-pairs lattice over $\widehat{L}$ given in Figure 6 together with Lemma 6. The bottom three complemented-pairs in Figure 6 also occur in Figure 5, and have therefore already been justified. We justify the remaining pairs as follows.

1. $\langle x \in done_r[r], x \in \bigcap_i done_i[i] \rangle$. Justified since r sends gossip messages to every other replica.
2. $\langle x \in \bigcap_i done_i[i], x \in stable_r[r] \rangle$. Justified since each i sends gossip messages to r.
3. $\langle x \in stable_r[r], x \in \bigcap_i stable_i[i] \rangle$. Justified since r sends gossip messages to every other replica.

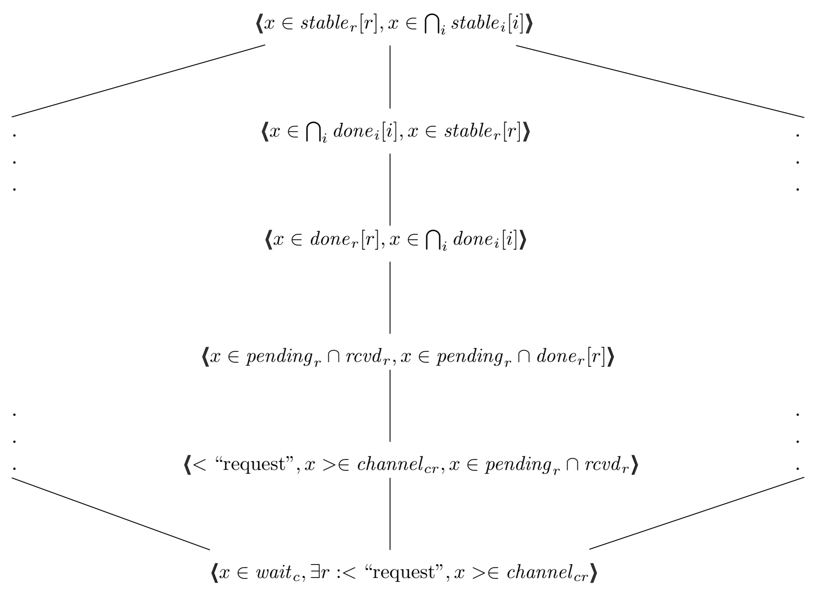
$\langle x \in stable_r[r], x \in \bigcap_i stable_i[i] \rangle$
$\langle x \in \bigcap_i done_i[i], x \in stable_r[r] \rangle$
$\langle x \in done_r[r], x \in \bigcap_i done_i[i] \rangle$
$\langle x \in pending_r \cap rcvd_r, x \in pending_r \cap done_r[r] \rangle$
$\langle <$ "request"$, x > \in channel_{cr}, x \in pending_r \cap rcvd_r \rangle$
$\langle x \in wait_c, \exists r :<$ "request"$, x > \in channel_{cr} \rangle$

Fig. 6 Complemented-pairs lattice that establishes $\langle x \in wait_c, x \in \bigcap_i stable_i[i] \rangle \in \widehat{L}$ ($c = client(x)$).

Since Definition 10 is now satisfied, we have established $(\mathit{ESDS\text{-}Alg} \parallel \mathit{Users}, L) \leq_{\ell F} (\mathit{ESDS\text{-}II} \parallel \mathit{Users}, \mathit{M\text{-}II})$. Hence, applying Theorem 3, we conclude $(\mathit{ESDS\text{-}Alg} \parallel \mathit{Users}, L) \sqsubseteq_{\ell} (\mathit{ESDS\text{-}II} \parallel \mathit{Users}, \mathit{M\text{-}II})$. Together with $(\mathit{ESDS\text{-}II} \parallel \mathit{Users}, \mathit{M\text{-}II}) \sqsubseteq_{\ell} (\mathit{ESDS\text{-}I} \parallel \mathit{Users}, \mathit{M\text{-}I})$ established above, we have $(\mathit{ESDS\text{-}Alg} \parallel \mathit{Users}, L) \sqsubseteq_{\ell} (\mathit{ESDS\text{-}I} \parallel \mathit{Users}, \mathit{M\text{-}I})$, as desired.

We have illustrated three levels of abstraction, and two liveness-preserving forward simulations, between the top and middle, and middle and bottom levels. It is straightforward to continue this process. For example, an actual implementation would not simply route a request to any replica, but would select the replica according to certain criteria, for example load balancing/performance [45], or distance from the client [49]. Thus, the front-ends and replicas would be refined to incorporate a load-balancing/anycast/replica (or mirror) location "service" which, given a request from a client c, assigns some replica r to service that request. We then map the complemented-pair $\langle x \in wait_c, \exists r :<$ "request"$, x > \in channel_{cr} \rangle$ into a pair at the next lower level which expresses the liveness of the service: the service eventually assigns some replica r to every request x. This pair could then be justified by constructing a lattice whose elements are the specified or derived liveness properties of the service.

7 Discussion

7.1 Alternative Choices for Specifying Liveness Properties

We have used the complemented-pairs acceptance condition to specify liveness properties. There are other acceptance conditions for finite automata over infinite strings that we could have chosen: Büchi, generalized-Büchi, Rabin, and Müller. We briefly discuss each in turn.

A Büchi condition is a single set Green of states, and the computation must contain an infinite number of states from Green. This can be expressed as a single complemented pair $\langle true, \mathsf{Green}\rangle$, and so is subsumed by complemented-pairs. A generalized-Büchi condition is a set $\{\mathsf{Green}_i \mid i \in \eta\}$ of sets of states, and for each Green_i, the computation should contain an infinite number of states from Green_i. This can be expressed as the set of complemented-pairs $\{\langle true, \mathsf{Green}_i\rangle \mid i \in \eta\}$ and so is also subsumed by complemented-pairs.

The Rabin condition is a set $\{\langle true, \mathsf{Green}_i\rangle \mid i \in \eta\}$ of pairs, however the acceptance condition is different. A computation α is accepted iff for some pair $\langle \mathsf{Red}_i, \mathsf{Green}_i\rangle$, α does not contain an infinite number of states in Red_i, and α does contain an infinite number of states in Green_i. This condition is a “disjunctive” one, it constrains a computation only with respect to any one of the pairs, not all of them at once. Since, in writing specifications, conjunction is far more useful than disjunction, i.e., we typically list some properties *all* of which must be satisfied, we feel that this condition would not be useful in practice.

The Müller condition is a set $\{\mathsf{Green}_i \mid i \in \eta\}$ of sets of states, and, the set of states that occur infinitely often along the computation should be exactly one of the Green_i. This condition is not very suitable for an infinite-state model, since it is possible (and indeed, often the case) that an infinite computation does not contain any particular state that recurs infinitely often, since the model usually contains unbounded data, such as integers, reals, sequences, or sets. Thus, the set of states *each of which* occurs infinitely often along the computation, is usually empty.

Finally, we consider the “temporal leads-to” property. Roughly, p leads-to q means that, whenever p holds, then q subsequently holds. In our framework, leads-to properties can be expressed and verified by using history variables. Let $flag_p$ be a boolean history variable that is initially false, is set whenever $p \wedge \neg q$ holds, and reset whenever q holds. Then, the complemented-pair $\langle flag_p, q\rangle$ expresses “p leads-to q.” Since $flag_p$ is not used to affect control flow, it does not need to be “implemented.” Thus, the issue of atomically detecting the values of p and q at run time and updating $flag_p$, does not arise.

7.2 Application to Fault-tolerance

Our method can be applied to the verification of *fault tolerance* properties. We consider situations in which the occurrence of a fault can cause the system to enter a “bad” state, i.e., one that is unreachable under normal execution [7]. Let *good* denote the set of states that are reachable under normal system execution from a start state, and let *fault* denote the set of states that result immediately after a fault occurs, i.e., the post-states of faults (the faults can occur in any state, good or bad). If follows that, under normal execution (no faults) only good states are reachable from good states.

We are interested in “nonmasking” fault-tolerance properties of the type: once faults stop occurring, the system will eventually recover to a good state (and therefore remain forever after in good states, since only good states are reachable from good states in the absence of faults). Expressed in temporal logic, this is $(\Diamond\Box\neg fault \Rightarrow \Box\Diamond good)$. This is logically equivalent to $\Box\Diamond(fault \vee good)$. We can express this as the complemented pair $\langle true, fault \vee good\rangle$.

Hence, the liveness condition $\langle true, fault \vee good \rangle$ defines the set of "live" executions to be either (1) those along which an infinite number of faults occur (in which case we have no obligation to recover to a good state) or (2) those along which an infinite number of good states occur. In the latter case, we may also assume that faults stop occurring, since the negation of this is covered by case (1). Since only good states are reachable from good states, it follows that there is some suffix consisting entirely of good states, and so the system has recovered.

Thus, "live" executions are those in which the system exhibits the desired fault-tolerance property. The trace of such an execution is then an "external fault-tolerant behavior."

We can now refine such nonmasking fault-tolerance properties, i.e., to establish that the external fault-tolerant behaviors of an implementation are included in those of the specification. Our framework thus can take the place of theories that are specialized to dealing with nonmasking fault-tolerance, e.g., [13], which we have shown is just a particular kind of liveness property.

7.3 Mechanization Of Our Method

Our method imposes the following proof obligations:

1. Devise an appropriate liveness-preserving simulation and check that it satisfies all of the conditions of its definition, i.e., Definition 10 or 11.
2. For each derived pair, devise a complemented-pairs lattice and check that it satisfies the conditions of Definition 14.

Given a live automaton (A, L), obligation 2 is concerned with showing that a derived pair p' is "semantically entailed" by the liveness condition L, i.e., that for every infinite execution α of A, if $(\forall p \in L : \alpha \models p)$, then $\alpha \models p'$. In the case that A is finite-state, the number of pairs in L is finite, and then the validity of the linear-time temporal logic formula $(\wedge p \in L : p) \Rightarrow p'$ is sufficient to establish this. Hence, we just check that $\neg((\wedge p \in L : p) \Rightarrow p')$ is not satisfiable, using a decision procedure for linear-time temporal logic.

In the infinite state case, obligations 1 and 2 can be formalized in a first-order assertion language with interpreted symbols. We refer the reader to [17,20] for details. The conditions can be verified using theorem provers such as PVS [47]. For lack of space, we omit an extended discussion of these issues, which can be found, for example, in [20]. That paper presents *normed simulations*, where the existence of a finite execution fragment at the abstract level that matches a concrete transition is replaced by the existence of either a single matching transition, or an internal transition that decreases a supplied norm (a function over a well-founded domain). It should be possible to extend the ideas in this paper to normed simulations. For example, if the concrete transition contains a Red state, then we require that, by the time that either the matching abstract transition has been generated, or the norm function has decreased to minimum, that a corresponding Red state has appeared at the abstract level. We leave the details to another occasion.

8 Expressive Completeness of Complemented-pairs Liveness Conditions

We now investigate the expressiveness of complemented-pairs: what are the live execution properties which can be expressed by complemented-pairs conditions? First, we make this notion precise.

Definition 15 Let A be an automaton and let φ be a live execution property for A. Then we say that a liveness condition L *expresses* φ if and only if (A, L) is a live automaton and $\mathit{lexecs}(A, L) = \varphi$.

The use of (complemented-pairs) liveness conditions to specify liveness means that the liveness of an execution depends only on the set of states which occur in that execution, and not on their ordering. This is necessary, to satisfy the machine closure condition, since ordering is a safety property: once an ordering is violated along a finite execution, no extension can then satisfy the ordering.

In Section 8.1, we show that, under some assumptions that are natural for infinite-state systems, that the generalized Büchi condition is expressively complete, i.e., it can express any live execution property. Since complemented pairs subsumes generalized Büchi, the result then carries over to our framework.

In Section 8.2, we show that complemented pairs are expressively complete if history variables can be used.

8.1 Relative Expressive Completeness of Complemented-pairs Liveness Conditions

In this section, we define a class of liveness properties which can be expressed using complemented pairs. Let $\alpha = s_0 a_1 s_1 \ldots$ be an infinite execution. Define $\mathit{states}(\alpha) = \{s \mid (\exists i \geq 0 : s = s_i)\}$, i.e., the set of states that occur at least once along α, and $\mathit{inf}(\alpha) = \{s \mid (\forall i \geq 0\ \exists j \geq i : s = s_j)\}$, i.e., the set of states that occur infinitely often along α. α is a *non-repeating* execution iff $\mathit{inf}(\alpha) = \emptyset$, otherwise α is a *repeating* execution. In infinite-state systems, it is often the case that the occurrence of "significant" events is permanently recorded by changes to the state. For instance, in the eventually-serializable data service of Section 6, the execution of every operation on the data results in a permanent record of that operation's unique identifier. Any database system which maintains logs is also an example of this. So is a real-time system in which clocks maintain the time, if we consider the passage of time to be a significant event. This large class of systems justifies the assumption that a particular state cannot repeat infinitely often along a live execution, i.e., that live executions are non-repeating, since we expect that significant events (e.g., operation execution, transaction commit, time passage) occur infinitely often along a live execution. Hence we wish to restrict attention to liveness properties consisting only of executions that are non-repeating. To do so, we must show that complemented-pairs can distinguish between repeating and non-repeating executions. This is straightforward. Let α be a non-repeating execution and α' a repeating execution. Let s be any state in $\mathit{inf}(\alpha')$, so that s occurs infinitely often along α'. Then $\alpha \models \langle s, \mathit{false} \rangle$ since $\alpha \models \neg\Box\Diamond s$, and $\alpha' \not\models \langle s, \mathit{false} \rangle$ since $\alpha' \models \Box\Diamond s$. Hence any pair of repeating and non-repeating executions can be distinguished by a single complemented pair.

Now consider two non-repeating executions γ and α. Since a generalized-Büchi condition depends only on the states which occur in an execution (and not on their ordering), we take it as reasonable that if α contains "as many" states as γ, and γ is live (w.r.t. generalized Büchi), then α should also be live. We call a liveness property that satisfies this condition *robust*. To formalize the notion of robust liveness property, we must precisely define the notion "α contains as many states as γ." An obvious candidate is $states(\gamma) \subseteq states(\alpha)$. However, we note that any finite change in $states(\gamma)$, $states(\alpha)$ does not affect the Büchi properties that γ, α satisfy, respectively. Hence, we use "$states(\gamma) - states(\alpha)$ is a finite set" as the required notion. This leads to:

Definition 16 (Robust Live Execution Property) Let φ be a live execution property for automaton A. Then, φ is *robust for A* if and only if the following hold:

1. Every execution in φ is non-repeating.
2. Let γ, α be infinite non-repeating executions of A. If $states(\gamma) - states(\alpha)$ is a finite set and $\gamma \in \varphi$, then $\alpha \in \varphi$.

Our robustness condition corresponds more closely to using a generalized-Büchi acceptance condition than a complemented-pairs acceptance condition (see Section 7.1 above). Since complemented-pairs subsume generalized-Büchi, this is still within our framework, and also allows for a simpler technical development. The definition of live trace properties corresponding to robust live execution properties is straightforward.

Definition 17 (Robust Live Trace Property) Let A be an automaton, and $\psi \subseteq traces(A)$. Then, ψ is a *robust live trace property* for A if and only if there exists a robust live execution property φ for A such that $\psi = traces(\varphi)$.

We now show that an infinite non-repeating execution outside φ can be distinguished from an execution in φ by means of a simple Büchi acceptance condition.

Proposition 3 *Let A be an automaton, and let φ be an arbitrary robust live execution property for A. Let γ, α be arbitrary infinite non-repeating executions of A such that $\gamma \in \varphi$ and $\alpha \notin \varphi$. Then there exists a set $\mathsf{G}_{\alpha,\gamma} \subseteq states(A)$ such that $\gamma \models \Box\Diamond \mathsf{G}_{\alpha,\gamma}$ and $\alpha \models \Box\neg \mathsf{G}_{\alpha,\gamma}$.*

Proof Suppose $states(\gamma) - states(\alpha)$ is finite. Then $\alpha \in \varphi$ by Definition 16. So, $states(\gamma) - states(\alpha)$ is infinite. Thus $\gamma \models \Box\Diamond(states(\gamma) - states(\alpha))$. Also, $\alpha \models \Box\neg(states(\gamma) - states(\alpha))$, by definition. So, letting $\mathsf{G}_{\alpha,\gamma} = states(\gamma) - states(\alpha)$ establishes the proposition.

We next show that an infinite non-repeating execution outside φ can be distinguished from every execution in φ by means of a simple Büchi acceptance condition.

Proposition 4 *Let A be an automaton, and let φ be an arbitrary robust live execution property for A. Let α be an arbitrary infinite non-repeating execution of A such that $\alpha \notin \varphi$. Then there exists a set $\mathsf{G}_\alpha \subseteq states(A)$ such that $\alpha \models \Box\neg \mathsf{G}_\alpha$ and $\forall \gamma \in \varphi : \gamma \models \Box\Diamond \mathsf{G}_\alpha$.*

Proof Let γ be an arbitrary execution in φ, and let $\mathsf{G}_{\alpha,\gamma}$ be the set given by Proposition 3 for α, γ. Then $\gamma \models \Box\Diamond \mathsf{G}_{\alpha,\gamma}$ and $\alpha \models \Box\neg \mathsf{G}_{\alpha,\gamma}$. Let $\mathsf{G}_\alpha = \bigcup_{\gamma \in \varphi} \mathsf{G}_{\alpha,\gamma}$. Then, $\forall \gamma \in \varphi : \gamma \models \Box\Diamond \mathsf{G}_\alpha$, since $\mathsf{G}_{\alpha,\gamma} \subseteq \mathsf{G}_\alpha$. Also, $\alpha \models \Box\neg \mathsf{G}_\alpha$ since $\alpha \models \Box\neg \mathsf{G}_{\alpha,\gamma}$ for every $\mathsf{G}_{\alpha,\gamma}$, $\gamma \in \varphi$.

We now present the relative completeness result: for any robust live execution property φ, there exists a complemented-pairs condition that is satisfied by exactly the executions in φ.

Theorem 4 (Relative Expressive Completeness of Complemented-pairs) *Let A be an automaton, and let φ be an arbitrary robust live execution property for A. Then there exists a complemented-pairs condition L over A such that $\varphi = lexecs(A, L)$.*

Proof Let $L = \{\langle true, \mathsf{G}_\alpha \rangle \mid \alpha \in nrexecs^\omega(A) - \varphi\} \cup \{\langle s, false \rangle \mid \alpha \in rexecs^\omega(A) \wedge s \in inf(\alpha)\}$, where G_α is as given in Proposition 4, $nrexecs^\omega(A)$ is the set of infinite non-repeating executions of A, and $rexecs^\omega(A)$ is the set of infinite repeating executions of A.

We consider all possible infinite executions of A. There are three cases.

Case 1: an arbitrary execution in φ. That is, some $\gamma \in \varphi$. By Proposition 4, $\gamma \models \Box\Diamond\mathsf{G}_\alpha$ for all $\alpha \in nrexecs^\omega(A) - \varphi$. Hence $\gamma \models \langle true, \mathsf{G}_\alpha \rangle$ for all $\alpha \in nrexecs^\omega(A) - \varphi$. By Definition 16, γ is non-repeating. Hence $\gamma \models \neg\Box\Diamond s$ for any single state s. Hence $\gamma \models \langle s, false \rangle$ for all $\alpha \in rexecs^\omega(A)$ and $s \in inf(\alpha)$.

Case 2: an arbitrary non-repeating execution not in φ. That is, some $\alpha \in nrexecs^\omega(A) - \varphi$. By Proposition 4, $\alpha \models \Box\neg\mathsf{G}_\alpha$. Hence $\alpha \not\models \langle true, \mathsf{G}_\alpha \rangle$.

Case 3: an arbitrary repeating execution not in φ. That is, some $\alpha \in rexecs^\omega(A) - \varphi$. Consider $\langle s, false \rangle$ for any $s \in inf(\alpha)$. Since $\alpha \models \Box\Diamond s$, we immediately have $\alpha \not\models \langle s, false \rangle$. (Note that $rexecs^\omega(A) - \varphi = rexecs^\omega(A)$ by Definition 16.)

The above three cases clearly exhaust all the possibilities for an infinite execution of A. We showed that every execution γ in φ satisfies every pair in L, and every execution α not in φ does not satisfy some pair in L. Hence $\varphi = lexecs(A, L)$.

Corollary 1 (Relative Expressive Completeness of Complemented-pairs) *Let A be an automaton, and let ψ be an arbitrary robust live trace property for A. Then there exists a complemented-pairs liveness condition L over A such that $traces(lexecs(A, L)) = \psi$.*

Proof Let ψ be an arbitrary robust live trace property for A. By Definition 17, there exists a robust live execution property φ for A such that $\psi = traces(\varphi)$. By Theorem 4, there exists a complemented-pairs condition L over A such that $\varphi = lexecs(A, L)$. Hence $traces(lexecs(A, L)) = \psi$ and we are done.

8.2 Expressive Completeness of Complemented-pairs for Liveness Properties of Forest Automata

An automaton A is a *forest automaton* iff for each reachable state s of A, there is exactly one (finite) execution of A with last state s. Thus, if α, α' are arbitrary different infinite executions of A, then they have only a finite number of states in common. Any automaton can be turned into a forest automaton by adding a history variable which records the execution up to the current state. While this is obviously impractical for a real implementation, such a variable is only needed for modeling and analysis purposes; it does not have to be implemented since it does not affect the actual execution of the automaton.[10]

Let α be an arbitrary infinite execution of A. Define $pair(\alpha) = \langle states(\alpha), \emptyset \rangle$.

Proposition 5 *Let A be a forest automaton. Then $\forall \alpha, \alpha' \in execs^\omega(A) : \alpha' \neq \alpha$ iff $\alpha' \models pair(\alpha)$.*

[10] The terms "ghost variable" and "auxiliary variable" have been used in the literature for this notion.

Proof Let α, α' be arbitrary elements of $\mathit{execs}^{\omega}(A)$. If $\alpha' \neq \alpha$, then $\alpha' \models \Diamond\Box\neg \mathit{states}(\alpha)$, since α, α' have only a finite number of states in common. Hence $\alpha' \not\models \Box\Diamond \mathit{states}(\alpha)$, and so $\alpha' \models \mathit{pair}(\alpha)$. If $\alpha' = \alpha$, then $\alpha' \models \Box\Diamond \mathit{states}(\alpha)$, and so $\alpha' \not\models \mathit{pair}(\alpha)$.

We show that, if φ is a live execution property for automaton A, then there exists a liveness condition which expresses φ, i.e. such that an execution satisfies every complemented-pair in the condition iff it is a member of φ.

Theorem 5 (Expressive Completeness of Complemented-pairs for Forest Automata) *Let A be a forest automaton, and let φ be an arbitrary live execution property for A. Then there exists a complemented-pairs liveness condition L over A such that $\mathit{lexecs}(A, L) = \varphi$.*

Proof If $\varphi = \mathit{execs}^{\omega}(A)$ then letting $L = \{\langle\!\langle \mathit{true}, \mathit{true} \rangle\!\rangle\}$ establishes the theorem. Hence we assume that φ is a proper subset of $\mathit{execs}^{\omega}(A)$ for the rest of the proof. Let $L = \{\mathit{pair}(\alpha) \mid \alpha \in \mathit{execs}^{\omega}(A) - \varphi\}$. We show that $\mathit{lexecs}(A, L) = \varphi$. The proof is by double-containment.
$\mathit{lexecs}(A, L) \subseteq \varphi$: Choose arbitrarily $\alpha' \in \mathit{lexecs}(A, L)$ and $\alpha \in \mathit{execs}^{\omega}(A) - \varphi$. Now $\mathit{lexecs}(A, L) \subseteq \mathit{execs}^{\omega}(A)$ by definition, and so $\alpha' \in \mathit{execs}^{\omega}(A)$. From the definition of L, we have $\alpha' \models \mathit{pair}(\alpha)$. Hence, by Proposition 5, $\alpha \neq \alpha'$. Since α was chosen arbitrarily from $\mathit{execs}^{\omega}(A) - \varphi$, we conclude $\alpha' \notin \mathit{execs}^{\omega}(A) - \varphi$. Hence $\alpha' \in \varphi$, since $\alpha' \in \mathit{execs}^{\omega}(A)$.
$\varphi \subseteq \mathit{lexecs}(A, L)$: Choose arbitrarily $\alpha' \in \varphi$ and $\alpha \in \mathit{execs}^{\omega}(A) - \varphi$. Hence $\alpha \neq \alpha'$. Hence, by Proposition 5, $\alpha' \models \mathit{pair}(\alpha)$. Since α was chosen arbitrarily from $\mathit{execs}^{\omega}(A) - \varphi$, we conclude, from the definition of L, that $\alpha' \in \mathit{lexecs}(A, L)$.

Corollary 2 (Expressive Completeness of Complemented-pairs for Forest Automata) *Let A be a forest automaton, and let ψ be an arbitrary live trace property for A. Then there exists a complemented-pairs liveness condition L over A such that $\mathit{traces}(\mathit{lexecs}(A, L)) = \psi$.*

Proof Let ψ be an arbitrary live trace property for A. By Definition 4, there exists a live execution property φ for A such that $\psi = \mathit{traces}(\varphi)$. By Theorem 5, there exists a liveness condition L over A such that $\mathit{lexecs}(A, L) = \varphi$. Hence there exists a liveness condition L over A such that $\mathit{traces}(\mathit{lexecs}(A, L)) = \psi$.

8.3 Discussion

The issue of the completeness of proof systems for concurrent programs has been treated extensively in the literature, In particular, Sistla [51], considers the problem of verifying that a concurrent program P satisfies a specification A. P is expressed as a transition system (S, R, S_0), where S is the set of program states, $S_0 \subseteq S$ is the set of initial program states, and $R \subseteq S \times S$ is the transition relation. A is a nondeterministic Büchi automaton that accepts/rejects the computations of P. An effective program is one where S, R, S_0 are recursive sets, and similarly for an effective Büchi automaton. The paper considers the problem PROG$-$SAT$-$BUCHI $= \{(P, A) \mid P$, A are both effective and A accepts all the computations of $P\}$, that is, the set of pairs (P, A) such that P satisfies A. It shows that PROG$-$SAT$-$BUCHI is $\mathbf{\Pi}_2^1$-complete. The paper then defines the notion of a $\mathbf{\Sigma}_2$ proof system, and shows that if a $\mathbf{\Sigma}_2$ proof system is complete for PROG−SAT−BUCHI, then PROG−SAT−BUCHI is in $\mathbf{\Sigma}^1_2$, contradicting the $\mathbf{\Pi}^1_2$-completeness result. Hence there does not exist a complete $\mathbf{\Sigma}_2$ proof system for PROG−SAT−BUCHI. Sistla also argues that it is unlikely that a complete and practical proof system for PROG−SAT−BUCHI will be developed, since all extant proof systems based on simulation relations are $\mathbf{\Sigma}_2$ proof systems.

The problem we consider generalizes PROG−SAT−BUCHI: let A be a concurrent program as in Sistla [51] with L consisting of the single pair $\langle true, true \rangle$, i.e., every execution of A is live, and let B be an automaton with set of pairs M consisting of the single pair $\langle true, \mathsf{Green} \rangle$, so that B is a nondeterministic Büchi automaton. Hence by the results of Sistla [51], complete and practical proof systems for refining liveness properties are unlikely to exist, and so the *relative* completeness results that we presented (i.e., we showed that our method is complete only in certain special cases) appear to be the best that can be hoped for.

9 Related Work

The use of an infinite number of complemented pairs was proposed by Vardi [56], which defines a recursive Streett automaton to be one whose transition relation is recursive, and whose complemented pairs are defined by recursive sets. Recursive Büchi automata are defined similarly. Recursive Wolper automata are those with a recursive transition relation and no acceptance conditions. Every infinite run of the Wolper automaton is accepting. The paper shows that Recursive Wolper, Büchi, and Street automata all accept the same set of languages, namely $\mathbf{\Sigma}^1_1$. In our approach, we make no restrictions on the set of complemented pairs. For example, we allow uncountable sets of pairs, which could be useful for specifications over uncountable domains, e.g., the reals.

The safety-liveness classification was first proposed in [28]. Formal characterizations of safety and liveness, variously based on Büchi automata, temporal logic, or the Borel hierarchy, were given in [2,38,53]. Many researchers have proposed deductive systems for proving properties of infinite-state reactive and distributed systems, including liveness properties, e.g., [3,29,30,39]. Some of the methods proposed to date incorporate diagrammatic techniques, similar in spirit to our complemented-pairs lattices. In particular, Owicki and Lamport [46] propose *proof lattices*, and Manna and Pnueli [37, 41] propose *proof diagrams*, both for establishing liveness properties of concurrent programs. In [42], Manna and Pnueli propose three different kinds of *verification diagrams*, two for safety properties, and one for liveness properties of the form $\Box(U \Rightarrow \Diamond V)$, where U, V are state-assertions, that is, temporal leads-to properties. Nodes in this diagram are labeled with state-assertions, and directed edges between nodes represent program transitions. Some of these edges correspond to "helpful" transitions, which are guaranteed to occur (using fairness) if execution enters their source node, and whose occurrence makes progress towards making V true. Browne et. al. [9] and Manna et. al. [36] present *generalized verification diagrams*, which can be used to establish arbitrary temporal properties of programs, including liveness properties. These are a particular kind of ω-automaton ("formula automata"). These methods relate a program, expressed in an operational notation, to a property expressed in temporal logic, i.e., they relate two artifacts expressed in very different notations. Thus, they cannot be used to refine liveness properties in a multi-stage stepwise refinement method that, starting with a high-level specification, expressed in a particular (operational) notation, constructs a sequence of artifacts, all expressed in the *same* notation, and each a refinement of the previous one, and ending with the detailed implementation.

Our complemented-pairs lattices relate a liveness property of an automaton, to a liveness property of a lower level automaton, i.e., the relationship is between two artifacts expressed in the same notation. This forms the basis for a multi-stage proof technique that refines high-level liveness properties down to the liveness properties of an implementation in several manageable steps (our use of "sublattices" in Section 6 is an example of this). Furthermore, each individual refinement step is itself decomposed into the tasks involved in constructing lattices and discharging the associated "verification conditions." We feel that this ability to decompose a liveness proof into multiple stages directly attacks the scalability problem, and is one of our main contributions. UNITY [10] provides a framework in which a subclass of general liveness properties, namely "leads-to" can be verified and refined. The approach is proof theoretic, and also relies on fairness. We showed in Section 7.1 above how to deal with leads-to properties in our framework. All of the aforementioned methods operate only at the level of executions, and do not provide a notion of external behavior, such as a set of traces.

Gawlick et. al. [18, 19] presents a proof method for liveness properties. In that paper, a liveness property of an automaton A is modeled as a subset L of the executions of A.[11] However, the method presented there imposes a proof obligation concerning the liveness of individual executions, without providing any rule or method for discharging this obligation. Specifically, in addition to establishing a simulation, we have to show that if an execution α of the implementation A corresponds to an execution α' of the specification B, and α is live (i.e., α is a member of the liveness property), then α' is also live[12]. Merely establishing a simulation between A and B is insufficient to show this, since the simulation relation makes no reference to the liveness conditions of A and B. The main concern in [18] is the interaction between liveness properties and parallel composition; a notion of "environment-freedom" is introduced which enables the use of compositional verification for liveness. The published version [19] omits the proof method.

Likewise, Jensen [24] presents simulation relations for proving liveness properties, and also requires that an "inclusion" condition be verified. A difference is that the live executions are exactly the fair executions, and so the inclusion property becomes: if an execution α of the implementation A corresponds to an execution α' of the specification B, and α is fair, then α' is also fair (Theorems 2.9 and 2.10 in [24]).

Sogaard-Andersen, Lynch, and Lampson [54] presents a similar method, with the main difference being that the liveness property is given by a linear temporal logic formula. Now, the proof obligation is that if an execution α of the implementation A corresponds to an execution α' of the specification B, and α satisfies the liveness formula for A, then α' satisfies the liveness formula for B.

Henzinger et. al. [22] presents various extensions of simulation that take fairness into account. Fairness is expressed using either Büchi or Streett (i.e., complemented-pairs) acceptance conditions. However, the fair simulation notions are defined using a game-theoretic semantics, and require a priori that fair executions of the concrete automaton have matching fair executions in the abstract automaton. There is no method of matching the Red and Green states in the concrete and abstract automata to assure

[11] L must satisfy the machine closure constraint of Definition 5.

[12] See [18], page 89.

fair trace containment. Also, the setting is finite state, and the paper concentrates on algorithms for checking fair simulation.

Alur and Henzinger [4] proposes the use of complemented-pairs acceptance conditions to define liveness properties. However it restricts the conditions to contain only a finite number of pairs. As our example in Section 6 shows, it is very convenient to be able to specify an infinite number of pairs—in this case, we were able to use two pairs for each operation x submitted to the data service, one pair to check for response, and the other to check for stabilization. It would be quite difficult to specify the liveness properties of the data service using only a finite number of pairs. If however, the system being considered is finite-state, then we remark that much of the work on temporal logic model checking seems applicable. For example, the algorithm of Emerson and Lei [14] for model checking under fairness assumptions can handle the complemented-pairs acceptance condition. While [4] gives rules for compositional and modular reasoning, it does not provide a method for refining liveness properties. As stated above, we believe this is a crucial aspect of a successful methodology for dealing with liveness. It should be clear that Figure 5 provides a very succinct presentation for the refinement of the liveness property expressed by $\langle\!\langle x \in \mathit{wait}, x \notin \mathit{wait} \rangle\!\rangle$, namely that every request eventually receives a response.

Our work is in the linear-time setting, where the external behavior is a set of traces. In the branching-time setting, the external behavior can be given as a "trace-tree" [22], i.e., a tree whose branches are traces. Our liveness-preserving simulation relations should imply an appropriate containment notion between "live-trace-trees," i.e., a tree whose branches are live traces. However we point out technical differences between our setting and [4, 22]: we abstract away states and internal actions to obtain traces, whereas in [4, 22] an execution is a sequence of states (actions are not named), and a trace is obtained by applying an "observation function" to each state along the execution.

Kesten, Pnueli, and Vardi [25, 26] present a method of *finitary abstraction*: construct a finite-state abstraction ("abstract system") of an infinite-state "concrete" system, and model check this abstraction for the required properties. The method deals with properties expressed in full linear time temporal logic, (and so handles both safety and liveness), and is complete, i.e., a suitable finite state abstraction can always be constructed. The semantics of the concrete system is given by a *Fair Discrete System* (FDS), which consists of (1) a finite set of typed system variables, containing the data and control state (the *concrete* variables), (2) a predicate giving the set of initial states, (3) a predicate giving the transition relation, (4) a *justice condition*; a finite set of predicates $\{J_1, ..., J_k\}$, where each J_i must hold infinitely often along a computation, and (5) a *compassion condition*, a finite set of pairs of predicates $\{< p_1, q_1 >, ..., < p_n, q_n >\}$; along a computation, if p_i holds infinitely often, then q_i must hold infinitely often. The justice and compassion conditions ensure that the concrete system satisfies liveness properties by restricting attention to "fair" computations. For a given concrete system, a finite-state abstract system is specified syntactically, by giving a set of abstract variables (with finite domains), and for each abstract variable, giving its value as an expression over the concrete variables. This implicitly defines a mapping from concrete to abstract states, and gives rise to two abstraction operators on concrete predicates: (1) a universal (contracting) abstraction, that holds in an abstract state iff the concrete predicate holds in all corresponding concrete states, and (2) an existential (expanding) abstraction, that holds in an abstract state iff the concrete predicate holds in some corresponding concrete state. The (concrete) temporal properties to be verified are abstracted by distributing these operators through temporal modalities (nexttime, until) and disjunction. Distribution through negation converts a universal abstraction into an existential one, and vice-versa. The abstract system is obtained by applying existential abstraction to the initial state predicate and each justice predicate. The transition relation is abstracted by "lifting" it to the abstract level using the definitions of the abstract variables in terms of the concrete variables. The compassion pairs $< p_i, q_i >$ are abstracted by applying universal abstraction to p_i and existential abstraction to q_i. A main result is that if the abstracted system satisfies the abstracted property, then the concrete system satisfies the concrete property. Another main result is that the method is complete: if the concrete system satisfies the property, then there exists a corresponding finite state abstract system and abstracted property such that the abstract system satisfies the abstract property. To obtain completeness, the concrete system must be "augmented" by composing it (synchronously) with a "ranking monitor," which tracks the difference in successive values of a variant function ("progress measure" in the paper) that decreases with progress towards satisfying the liveness property, and is defined over a well-founded domain. The reason for incompleteness of the unaugmented method is liveness properties. A major difference with our approach is that the number of complemented pairs is finite, whereas we allow an infinite set. Furthermore, the abstract system in our approach is not necessarily finite state. Verification in our approach is by manually devising a liveness-preserving simulation relation, and the needed complemented pairs lattices, and then checking the conditions in the corresponding definitions, possibly with mechanization via theorem proving (see Section 7.3). Verification in [25,26] is by manually devising the finitary abstraction mapping and the ranking monitors, and then model-checking the resulting abstracted system against the abstracted property. There is no method for deriving a liveness property at one level from other liveness properties at the same level, like our complemented-pairs lattices provide.

In [55], a method of abstraction based on Galois theory is presented. This is based on extensions of the framework of abstract interpretation [11] to temporal properties. Again, there are two abstraction notions: under-approximation and over-approximation. In [12], the interaction between abstraction and model checking under fairness is discussed. It is pointed out that abstraction really requires three-valued logic, since, e.g., a proposition that is true in one concrete state and false in another has "unknown" value in an abstract state that represents both concrete states. To handle fairness properly, two abstractions of the transition relation are introduced, called the free and constrained transition relations.

10 Conclusions and Further Work

We have presented two liveness-preserving simulation relations that allow us to refine the liveness properties of infinite-state distributed systems. Our method for refining liveness requires reasoning only over individual states and finite execution fragments, rather than reasoning over entire executions. We believe that the use of simulation-based refinement together with complemented-pairs lattices for expressing and combining liveness properties provides a powerful and general framework for refining liveness properties. In particular, our approach facilitates the decomposition of the refinement task at each level into simpler subtasks: devise the liveness-preserving simulation relation, and devise the complemented-pairs lattices. Since the lattices are a kind of diagram, they also facilitate the decomposition of proofs and the separation of concerns, which contributes to scalability of the method.

The general approach and techniques used in this paper do not depend intimately on the particular automaton model that we used. Thus, for example, our approach can be applied to labeled transition systems, which are used to define operational semantics for process algebras such as Algebra of Communicating Processes [8], Communicating Sequential Processes [23], Calculus of Communicating Systems [43], and the π-calculus [44]. Our approach can also be extended in a straightforward way to formalisms with unlabeled actions, such as (finite or infinite) Kripke structures, since the fact that actions are named is not used in any essential way, it just contributes to the "matching" condition in simulation relations, and to the definition of external behavior (trace).

We showed that the Streett acceptance condition (generalized to arbitrary cardinality) is expressive enough to define any liveness property, provided that it satisfies a notion of robustness, or provided that history variables can be used.

Simulation relations as a proof method for refinement have been widely studied. One major impediment to their widespread adoption in practice is the absence of efficient methodologies for establishing simulation relations. Doing so usually requires long proofs, with many invariants, etc. Some of the ideas in this paper may be applicable to decomposing and simplifying the task of establishing simulation relations in the first place. For example, it may be possible to apply our approach to refining the invariants that are used in such proofs. Another potential application is to models of computation for dynamic [6], real-time [32], hybrid [33], and probabilistic [50] systems. For example, a real-time analogue of a complemented-pair condition would be: if a Red state occurs, then a Green state must occur within t time units. A complemented-pairs lattice that refines a complemented-pair would then have to satisfy, in addition to the current requirements of Definition 14, a condition for the time bounds: every path from the bottom element to the top element should have a "total" time bound matching the pair being refined. In [6], we present an automata-theoretic model for dynamic computation, in which individual processes (automata) that constitute a system can be created and destroyed, and can dynamically change their action signature. Since the techniques of this paper assume only a generic automaton structure, they are applicable to the model of [6]. Combining these two pieces of work will result in a comprehensive method for verifying the liveness properties of dynamic systems.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing.*, 2(3):117–126, 1987.
3. B. Alpern and F. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Program. Lang. Syst.*, 11(1):147–167, Jan. 1989.
4. R. Alur and T. A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 166–179. Springer-Verlag, 1995.
5. P. C. Attie. On the refinement of liveness properties of distributed systems. Technical report, Department of Computer Science, American University of Beirut, Feb. 2008. available at http://arxiv.org/PS_cache/arxiv/pdf/0801/0801.0949v1.pdf.
6. P. C. Attie and N.A. Lynch. Dynamic input/output automata: a formal model for dynamic systems (extended abstract). In *CONCUR'01: 12th International Conference on Concurrency Theory*, LNCS. Springer-Verlag, Aug. 2001.

7. P.C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst.*, 26(1):125–185, Jan. 2004.
8. J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
9. A. Browne, Z. Manna, and H. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026, pages 484–498. Springer-Verlag LNCS, Dec. 1995.
10. K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Reading, Mass., 1988.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4'th Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.
12. D. Dams, R. Gerth, and O. Grumberg. Fair model checking of abstractions. In *Proceedings of the Workshop on Verification and Computational Logic (VCL'2000)*, University of Southampton, July 2000. Springer-Verlag.
13. M. Demirbas and A. Arora. Convergence refinement. In *International conference on distributed computing systems*, Vienna,Austria, July 2002.
14. E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. In *12'th Ann. ACM Symp. on Principles of Programming Languages*, pages 84–96, New Orleans, Louisiana, Jan. 1985. ACM Press.
15. A. Fekete, D. Gupta, V. Luchango, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113–156, 1999. Conference version appears in ACM Symposium on Principles of Distributed Computing, 1996.
16. N. Francez. *Fairness*. Springer-Verlag, New York, 1986.
17. S. J. Garland and N. A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.
18. R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT Laboratory for Computer Science, Boston, Mass., Nov. 1993.
19. R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, Mar. 1998.
20. D. Griffioen and F. Vaandrager. A theory of normed simulations. *ACM Transactions on Computational Logic*, 5(4):577–610, 2004.
21. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.
22. T.A. Henzinger, O. Kupferman, and S.K. Rajamani. Fair simulation. In A. mazurkiewicz and J. Wonkowski, editors, *CONCUR'97: Eighth International Conference on Concurrency Theory*, Lecture Notes in Computer Science 939, pages 273–287, Warsaw, Poland, July 1997. Springer-Verlag.
23. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.
24. H.E. Jensen. *Abstraction-based Verification of Distributed Systems*. PhD thesis, Institute for Computer Science, Aalborg University, June 1999.
25. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
26. Y. Kesten, A. Pnueli, and M. Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. *Journal of Computer and System Sciences*, 62(4):668–690, 2001.
27. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 1992.
28. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.
29. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.
30. L Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.
31. B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811 – 1841, Nov. 1994.
32. N.A. Lynch, R. Segala, F.Vaandrager, and D.K. Kaynar. Timed I/O automata. In preparation, 2003.

33. N.A. Lynch, R. Segala, and F. Vaandraager. Hybrid I/O automata. Technical Report MIT-LCS-TR-827d, MIT Laboratory for Computer Science, Cambridge, MA 02139, Jan. 2003. To appear in Information and Computation.
34. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report CWI-Quarterly, 2(3):219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, Sept. 1989.
35. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations — part I: Untimed systems. *Information and Computation*, 121(2):214–233, sep 1995.
36. Z. Manna, A. Browne, H. Sipma, and T. Uribe. Visual abstraction for temporal verification. In *AMAST'98*, volume 1548, pages 28–41. Springer-Verlag LNCS, 1998.
37. Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *ACM Principles of Programming Languages*, Austin, Texas, Jan. 1983.
38. Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *9'th podc*, pages 377–408, Quebec, Canada, Aug. 1990.
39. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science Journal*, 83(1):97–130, 1991.
40. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.
41. Z. Manna and A. Pnueli. A temporal proof methodology for reactive systems. In *Program Design Calculi, volume 118 of NATO ASI Series, Series F: Computer and System Sciences*, pages 287–323. Springer-Verlag, 1993.
42. Z. Manna and A. Pnueli. Temporal verification diagrams. In *International Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 789, pages 726–765. Springer-Verlag, 1994.
43. R. Milner. *Communication and Concurrency.* Prentice-Hall, Hemel Hempstead, U.K., 1989.
44. R. Milner. *Communicating and mobile systems: the π-calculus.* Addison-Wesley, Reading, Mass., 1999.
45. A. Myers, P. Dinda, and H. Zhang. Performance characteristics of mirror servers on the internet. In *IEEE INFOCOM*, 1999.
46. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, July 1982.
47. S. Owre, N. Shankar, and J. Rushby. Pvs: A prototype verification system. In *Proceedings CADE 11*, Saratoga Springs, NY, jun 1992.
48. A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Press, 1977.
49. K. Pruhs and B. Kalyanasundaram. The online transportation problem. *The SIAM Journal on Discrete Mathematics*, 13(3):370–383, 2000.
50. R. Segala. A compositional trace-based semantics for probabilistic automata. In Insup Lee and Scott A. Smolka, editors, *CONCUR'95: Concurrency Theory (6th International Conference)*, volume 962 of *LNCS*, pages 234–248. Springer-Verlag, 1995.
51. A. P. Sistla. On verifying that a concerrent program satisfies a non-deterministic specification. *Inf. Process. Lett.*, 32:17–23, jul 1989.
52. A. P. Sistla. Proving correctness with respect to non-deterministic safety specifications. *Inf. Process. Lett.*, 39:45–49, jul 1991.
53. A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects in Computing*, 6:495–511, 1994.
54. J.F. Sogaard-Andersen, N.A. Lynch, and B.W. Lampson. Correctness of comunication protocols: a case study. Technical Report MIT/LCS/TR-589, MIT Laboratory for Computer Science, Boston, Mass., Nov. 1993.
55. T. E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems.* PhD thesis, Computer Science Department, Stanford University, Dec. 1998. Technical Report STAN-CS-TR-99-1618.
56. M. Y. Vardi. Verification of concurrent programs — the automata theoretic framework. *Annals of pure applied logic*, 51:79–98, 1991.

A Simulation Relations

We present here two simulation relations, using the definitions of [35].

Definition 18 (Forward Simulation) Let A and B be automata with the same external actions. A *forward simulation* from A to B is a relation f over $states(A) \times states(B)$ that satisfies:

1. If $s \in start(A)$, then $f[s] \cap start(B) \neq \emptyset$.
2. If $s \xrightarrow{a}_A s'$ and $u \in f[s]$, then there exists a finite execution fragment α of B such that $fstate(\alpha) = u$, $lstate(\alpha) \in f[s']$, and $trace(\alpha) = trace(a)$.

Simulation based proof methods typically use *invariants* to restrict the steps that have to be considered. An invariant of an automaton is a predicate that holds in all of its reachable states, or alternatively, is a superset of the reachable states.

Definition 19 (Forward Simulation w.r.t. Invariants) Let A and B be automata with the same external actions and with invariants I_A, I_B, respectively. A *forward simulation* from A to B with respect to I_A and I_B is a relation f over $states(A) \times states(B)$ that satisfies:

1. If $s \in start(A)$, then $f[s] \cap start(B) \neq \emptyset$.
2. If $s \xrightarrow{a}_A s'$, $s \in I_A$, and $u \in f[s] \cap I_B$, then there exists a finite execution fragment α of B such that $fstate(\alpha) = u$, $lstate(\alpha) \in f[s']$, and $trace(\alpha) = trace(a)$.

We write $A \leq_F B$ if there exists a forward simulation from A to B w.r.t. some invariants, and $A \leq_F B$ via f if f is a forward simulation from A to B w.r.t. some invariants.

Definition 20 (Backward Simulation w.r.t. Invariants) Let A and B be automata with the same external actions and with invariants I_A, I_B, respectively. A *backward simulation* from A to B with respect to I_A and I_B is a relation b over $states(A) \times states(B)$ that satisfies:

1. If $s \in I_A$, then $b[s] \cap I_B \neq \emptyset$.
2. If $s \in start(A)$, then $b[s] \cap I_B \subseteq start(B)$.
3. If $s \xrightarrow{a}_A s'$, $s \in I_A$, and $u' \in b[s'] \cap I_B$, then there exists a finite execution fragment α of B such that $fstate(\alpha) \in b[s] \cap I_B$, $lstate(\alpha) = u'$, and $trace(\alpha) = trace(a)$.

A backward simulation b w.r.t. invariants is *image-finite* iff for each $s \in states(A)$, $b[s]$ is a finite set. We write $A \leq_B B$ if there exists a backward simulation from A to B w.r.t. some invariants, and $A \leq_B B$ via b if b is a backward simulation from A to B w.r.t. some invariants. If the backward simulation is image-finite, then we write $A \leq_{iB} B$, $A \leq_{iB} B$ via b, respectively.

B Linear-time Temporal Logic

We define the syntax and semantics of the temporal logic that we use as follows. This is essentially linear-time temporal logic without the until and nexttime operators.

Definition 21 (Syntax of Linear-time Temporal Logic) The syntax of a linear-time temporal logic formula is given inductively as follows, where f, g are sub-formulae, and U is a set of states (which defines a state-assertion):

- Each of U, $f \wedge g$ and $\neg f$ is a formula
- $\Box f$ is a formula which intuitively means that f holds in every state of the execution being considered
- $\Diamond f$ is a formula which intuitively means that f holds in some state of the execution being considered

Formally, we define the semantics of linear-time temporal logic formulae with respect to an infinite execution, that is, an infinite sequence of states.

Definition 22 (Semantics of Linear-time Temporal Logic) We use the usual notation to indicate truth: $\alpha \models f$ means that f is true of execution α. We define $\models$ inductively, where $\alpha = s_0 s_1 s_2 \ldots$ is an infinite sequence of states, and $\alpha^i = s_i s_{i+1} \ldots$ is the suffix of α starting in s_i.

$\alpha \models U$	iff	$s_0 \in U$
$\alpha \models \neg f$	iff	it is not the case that $\alpha \models f$
$\alpha \models f \wedge g$	iff	$\alpha \models f$ and $\alpha \models g$
$\alpha \models \Box f$	iff	for all $i \geq 0$, $\alpha^i \models f$
$\alpha \models \Diamond f$	iff	for some $i \geq 0$, $\alpha^i \models f$

In particular, $\alpha \models \Box\Diamond f$ means that $\alpha^i \models f$ for an infinite number of values of i.

C I/O Automaton Code for the ESDS Example, from [15]

I/O Automaton *Users*

Signature

Input:
  response(x, v), where $x \in \mathcal{O}$ and $v \in V$
Output:
  request(x), where $x \in \mathcal{O}$

State

requested, a subset of $\mathcal{O}$, initially empty

Actions

```
Output request$(x)$
Pre: $x.id \notin requested.id$
     $x.prev \subseteq requested.id$
Eff: $requested \leftarrow requested \cup \{x\}$

Input response$(x, v)$
Eff: None
```

Fig. 7 The Users Automaton

I/O Automaton *ESDS-I*

Signature

Input:
  request(x), where $x \in \mathcal{O}$
Output:
  response(x, v), where $x \in \mathcal{O}$ and $v \in V$
Internal:
  enter(x, *new-po*), where $x \in \mathcal{O}$ and *new-po* is a strict partial order on $\mathcal{I}$
  stabilize(x), where $x \in \mathcal{O}$
  calculate(x, v), where $x \in \mathcal{O}$ and $v \in V$
  add_constraints(*new-po*), where *new-po* is a partial order on $\mathcal{I}$

State

wait, a subset of $\mathcal{O}$, initially empty; the operations requested but not yet responded to
rept, a subset of $\mathcal{O} \times V$, initially empty; operations and responses that may be returned to clients
ops, a subset of $\mathcal{O}$, initially empty; the set of all operations that have ever been entered
po, a partial order on $\mathcal{I}$, initially empty; constraints on the order operations in *ops* are applied
stabilized, a subset of $\mathcal{O}$, initially empty; the set of stable operations

Actions

```
Input request($x$)
Eff: $wait \leftarrow wait \cup \{x\}$

Internal enter($x$, $new\text{-}po$)
Pre: $x \in wait$
     $x \notin ops$
     $x.prev \subseteq ops.id$
     $span(new\text{-}po) \subseteq ops.id \cup \{x.id\}$
     $po \subseteq new\text{-}po$
     $CSC(\{x\}) \subseteq new\text{-}po$
     $\{(y.id, x.id) : y \in stabilized\} \subseteq new\text{-}po$
Eff: $ops \leftarrow ops \cup \{x\}$
     $po \leftarrow new\text{-}po$

Internal add_constraints($new\text{-}po$)
Pre: $span(new\text{-}po) \subseteq ops.id$
     $po \subseteq new\text{-}po$
Eff: $po \leftarrow new\text{-}po$

Internal stabilize($x$)
Pre: $x \in ops$
     $x \notin stabilized$
     $\forall y \in ops,\ y \preceq_{po} x \vee x \preceq_{po} y$
     $ops|_{\prec_{po} x} \subseteq stabilized$
Eff: $stabilized \leftarrow stabilized \cup \{x\}$

Internal calculate($x, v$)
Pre: $x \in ops$
     $x.strict \Rightarrow x \in stabilized$
     $v \in valset(x, ops, \prec_{po})$
Eff: if $x \in wait$ then $rept \leftarrow rept \cup \{(x, v)\}$

Output response($x, v$)
Pre: $(x, v) \in rept$
     $x \in wait$
Eff: $wait \leftarrow wait - \{x\}$
     $rept \leftarrow rept - \{(x, v') : (x, v') \in rept\}$
```

Fig. 8 The Specification ESDS-I

```
Internal enter($x$, $new\text{-}po$)
Pre: $x \in wait$
     $x.prev \subseteq ops.id$
     $span(new\text{-}po) \subseteq ops.id \cup \{x.id\}$
     $po \subseteq new\text{-}po$
     $CSC(\{x\}) \subseteq new\text{-}po$
     $\{(y.id, x.id) : y \in stabilized\} \subseteq new\text{-}po$
Eff: $ops \leftarrow ops \cup \{x\}$
     $po \leftarrow new\text{-}po$

Internal stabilize($x$)
Pre: $x \in ops$
     $\forall y \in ops,\ y \preceq_{po} x \vee x \preceq_{po} y$
     $\prec_{po}$ totally orders $ops|_{\prec_{po} x}$
Eff: $stabilized \leftarrow stabilized \cup \{x\}$
```

Fig. 9 The Specification ESDS-II. Only differences with ESDS-I are shown.

I/O Automaton *Frontend*(c)

Signature

Input:
  $\mathsf{request}(x)$, where $x \in \mathcal{O}$ and $c = client(x)$
  $\mathsf{receive}_{rc}(m)$, where r is a replica and $m \in \mathcal{M}_{resp}$
Output:
  $\mathsf{response}(x, v)$, where $x \in \mathcal{O}$, $c = client(x)$, and $v \in V$
  $\mathsf{send}_{cr}(m)$, where r is a replica and $m \in \mathcal{M}_{req}$

State

$wait_c$, a subset of $\mathcal{O}$, initially empty
$rept_c$, a subset of $\mathcal{O} \times V$, initially empty

Actions

```
Input request(x)
Eff:  $wait_c \leftarrow wait_c \cup \{x\}$

Output send_{cr}(⟨"request", x⟩)
Pre:  $x \in wait_c$
Eff:  None

Input receive_{rc}(⟨"response", x, v⟩)
Eff:  if $x \in wait_c$ then $rept_c \leftarrow rept_c \cup \{(x, v)\}$

Output response(x, v)
Pre:  $(x, v) \in rept_c$
      $x \in wait_c$
Eff:  $wait_c \leftarrow wait_c - \{x\}$
      $rept_c \leftarrow rept_c - \{(x, v') : (x, v') \in rept_c\}$
```

Fig. 10 The Automaton for the front end of client c

I/O Automaton *Replica*(r)

Signature

Input:

- $\mathsf{receive}_{cr}(m)$, where c is a client and $m \in \mathcal{M}_{req}$
- $\mathsf{receive}_{r'r}(m)$, where $r' \neq r$ is a replica and $m \in \mathcal{M}_{gossip}$

Output:

- $\mathsf{send}_{rc}(m)$, where c is a client and $m \in \mathcal{M}_{resp}$
- $\mathsf{send}_{rr'}(m)$, where $r' \neq r$ is a replica and $m \in \mathcal{M}_{gossip}$

Internal:

- $\mathsf{do\_it}_r(x, l)$, where $x \in \mathcal{O}$ and $l \in \mathcal{L}_r$

State

$pending_r$, a subset of $\mathcal{O}$, initially empty; the messages that require a response
$rcvd_r$, a subset of $\mathcal{O}$, initially empty; the operations that have been received
$done_r[i]$ for each replica i, a subset of $\mathcal{O}$, initially empty; the operations r knows are done at i
$stable_r[i]$ for each replica i, a subset of $\mathcal{O}$, initially empty; the operations r knows are stable at i
$label_r : \mathcal{I} \rightarrow \mathcal{L} \cup \{\infty\}$, initially all ∞; the minimum label r has seen for $id \in \mathcal{I}$
Derived variable: $lc_r = \{(id, id') : label_r(id) < label_r(id')\}$, a strict partial order on $\mathcal{I}$; the local constraints at r

Actions

```
Input receive_cr($\langle$"request", $x\rangle$)
Eff: $pending_r \leftarrow pending_r \cup \{x\}$
     $rcvd_r \leftarrow rcvd_r \cup \{x\}$

Internal do_it_r($x, l$)
Pre: $x \in rcvd_r - done_r[r]$
     $x.prev \subseteq done_r[r].id$
     $l > label_r(y.id)$ for all $y \in done_r[r]$
Eff: $done_r[r] \leftarrow done_r[r] \cup \{x\}$
     $label_r(x.id) \leftarrow l$

Output send_rc($\langle$"response", $x, v\rangle$)
Pre: $x \in pending_r \cap done_r[r]$
     $x.strict \Rightarrow x \in \bigcap_i stable_r[i]$
     $v \in valset(x, done_r[r], \prec_{lc_r})$
     $c = client(x)$
Eff: $pending_r \leftarrow pending_r - \{x\}$

Output send_rr'($\langle$"gossip", $R, D, L, S\rangle$)
Pre: $R = rcvd_r$; $D = done_r[r]$;
     $L = label_r$; $S = stable_r[r]$

Input receive_r'r($\langle$"gossip", $R, D, L, S\rangle$)
Eff: $rcvd_r \leftarrow rcvd_r \cup R$
     $done_r[r'] \leftarrow done_r[r'] \cup D \cup S$
     $done_r[r] \leftarrow done_r[r] \cup D \cup S$
     $done_r[i] \leftarrow done_r[i] \cup S$ for all $i \neq r, r'$
     $label_r \leftarrow \min(label_r, L)$
     $stable_r[r'] \leftarrow stable_r[r'] \cup S$
     $stable_r[r] \leftarrow stable_r[r] \cup S \cup (\bigcap_i done_r[i])$
```

Fig. 11 Automaton for replica r

I/O Automaton *Channel*$(i, j, \mathcal{M})$

Signature

Input:
  $\mathsf{send}_{ij}(m)$, where $m \in \mathcal{M}$
Output:
  $\mathsf{receive}_{ij}(m)$, where $m \in \mathcal{M}$

State

$channel_{ij}$, a multiset of messages, (taken from $\mathcal{M}$), initially empty

Actions

Input $\mathsf{send}_{ij}(m)$
Eff: $channel_{ij} \leftarrow channel_{ij} \cup \{m\}$

Output $\mathsf{receive}_{ij}(m)$
Pre: $m \in channel_{ij}$
Eff: $channel_{ij} \leftarrow channel_{ij} - \{m\}$

Fig. 12 The Channel Automaton