SAND2020-10243C

Scalable Termination Detection for Distributed Actor Systems

Dan Plyukhin
University of Illinois at Urbana-Champaign, Urbana, IL, USA
daniilp2@illinois.edu

Gul Agha
University of Illinois at Urbana-Champaign, Urbana, IL, USA
agha@illinois.edu

Abstract

Automatic *garbage collection* (GC) prevents certain kinds of bugs and reduces programming overhead. GC techniques for sequential programs are based on *reachability analysis*. However, testing reachability from a root set is inadequate for determining whether an *actor* is garbage because an unreachable actor may send a message to a reachable actor. Instead, it is sufficient to check *termination* (sometimes also called *quiescence*): an actor is terminated if it is not currently processing a message and cannot receive a message in the future. Moreover, many actor frameworks provide all actors with access to file I/O or external storage; without inspecting an actor's internal code, it is necessary to check that the actor has terminated to ensure that it may be garbage collected in these frameworks. Previous algorithms to detect actor garbage require coordination mechanisms such as causal message delivery or nonlocal monitoring of actors for mutation. Such coordination mechanisms adversely affect concurrency and are therefore expensive in distributed systems. We present a low-overhead *reference listing* technique (called *DRL*) for termination detection in actor systems. DRL is based on asynchronous local snapshots and message-passing between actors. This enables a decentralized implementation and transient network partition tolerance. The paper provides a formal description of DRL, shows that all actors identified as garbage have indeed terminated (safety), and that all terminated actors–under certain reasonable assumptions–will eventually be identified (liveness).

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms; Software and its engineering → Garbage collection

Keywords and phrases actors, concurrency, termination detection, quiescence detection, garbage collection, distributed systems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2020.11

Funding This work was supported in part by the National Science Foundation under Grant No. SHF 1617401, and in part by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Acknowledgements We would like to thank Dipayan Mukherjee, Atul Sandur, Charles Kuch, Jerry Wu, and the anonymous referees for providing valuable feedback in earlier versions of this work.

1 Introduction

The actor model [1, 2] is a foundational model of concurrency that has been widely adopted for its scalability: for example, actor languages have been used to implement services at PayPal [19], Discord [27], and in the United Kingdom's National Health Service database [18]. In the actor model, stateful processes known as *actors* execute concurrently and communicate by sending asynchronous messages to other actors, provided they have a *reference* (also called

a *mail address* or *address* in the literature) to the recipient. Actors can also spawn new actors. An actor is said to be *garbage* if it can be destroyed without affecting the system's observable behavior.

Although a number of algorithms for automatic actor GC have been proposed [10, 13, 24, 25, 28, 30], actor languages and frameworks currently popular in industry (such as Akka [4], Erlang [5], and Orleans [8]) require that programmers garbage collect actors manually. We believe this is because the algorithms proposed thus far are too expensive to implement in distributed systems. In order to find applicability in real-world actor runtimes, we argue that a GC algorithm should satisfy the following properties:

1. (*Low latency*) GC should not restrict concurrency in the application.
2. (*High throughput*) GC should not impose significant space or message overhead.
3. (*Scalability*) GC should scale with the number of actors and nodes in the system.

To the best of our knowledge, no previous algorithm satisfies all three constraints. The first requirement precludes any global synchronization between actors, a "stop-the-world" step, or a requirement for causal order delivery of all messages. The second requirement means that the number of additional "control" messages imposed by the algorithm should be minimal. The third requirement precludes algorithms based on global snapshots, since taking a global snapshot of a system with a large number of nodes is infeasible.

To address these goals, we have developed a garbage collection technique called *DRL* for *Deferred Reference Listing*. The primary advantage of DRL is that it is decentralized and incremental: local garbage can be collected at one node without communicating with other nodes. Garbage collection can be performed concurrently with the application and imposes no message ordering constraints. We also expect DRL to be reasonably efficient in practice, since it does not require many additional messages or significant actor-local computation.

DRL works as follows. The *communication protocol* (Section 4) tracks information, such as references and message counts, and stores it in each actor's state. Actors periodically send out copies of their local state (called *snapshots*) to be stored at one or more designated *snapshot aggregator* actors. Each aggregator periodically searches its local store to find a subset of snapshots representing terminated actors (Section 6). Once an actor is determined to have terminated, it can be garbage collected by, for example, sending it a *self-destruct* message. Note that our termination detection algorithm itself is *location transparent*.

Since DRL is defined on top of the actor model, it is oblivious to details of a particular implementation (such as how sequential computations are represented). Our technique is therefore applicable to any actor framework and can be implemented as a library. Moreover, it can also be applied to open systems, allowing a garbage-collected actor subsystem to interoperate with an external actor system.

The outline of the paper is as follows. We provide a characterization of actor garbage in Section 2 and discuss related work in Section 3. We then provide a specification of the DRL protocol in Section 4. In Section 5, we describe a key property of DRL called the *Chain Lemma*. This lemma allows us to prove the safety and liveness properties, which are stated in Section 6. We then conclude in Section 7 with some discussion of future work and how DRL may be used in practice. To conserve space, all proofs have been relegated to the Appendix.

2 Preliminaries

An actor can only receive a message when it is *idle*. Upon receiving a message, it becomes *busy*. A busy actor can perform an unbounded sequence of *actions* before becoming idle. In [3], an action may be to spawn an actor, send a message, or perform a (local) computation. We will also assume that actors can perform effects, such as file I/O. The actions an actor performs in response to a message are dictated by its application-level code, called a *behavior*.

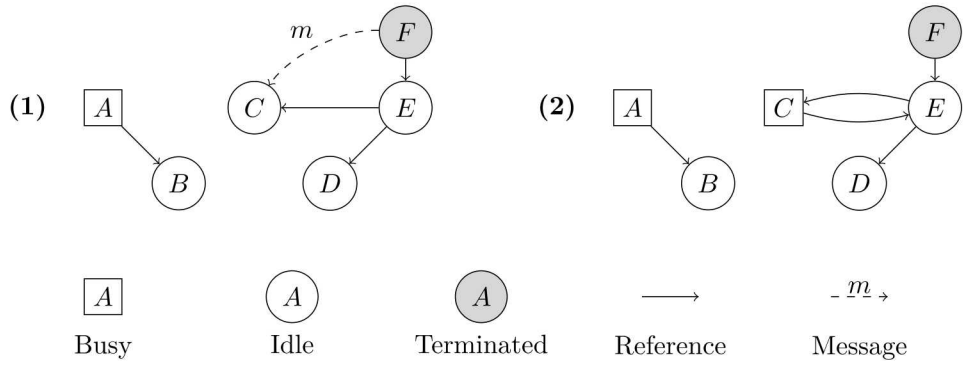

Figure 1 A simple actor system. The first configuration leads to the second after C receives the message m, which contains a reference to E. Notice that an actor can send a message and "forget" its reference to the recipient before the message is delivered, as is the case for actor F. In both configurations, E is a potential acquaintance of C, and D is potentially reachable from C. The only terminated actor is F because all other actors are potentially reachable from unblocked actors.

Actors can also receive messages from *external* actors (such as the user) by becoming *receptionists*. An actor A becomes a receptionist when its address is exposed to an external actor. Subsequently, any external actor can potentially obtain A's address and send it a message. It is not possible for an actor system to determine when all external actors have "forgotten" a receptionist's address. We will therefore assume that an actor can never cease to be a receptionist once its address has been exposed.

An actor is said to be garbage if it can be destroyed without affecting the system's observable behavior. However, without analyzing an actor's code, it is not possible to know whether it will have an effect when it receives a message. We will therefore restrict our attention to actors that can be guaranteed to be garbage without inspecting their behavior. According to this more conservative definition, any actor that might receive a message in the future should not be garbage collected because it could, for instance, write to a log file when it becomes busy. Conversely, any actor that is guaranteed to remain idle indefinitely can safely be garbage collected because it will never have any effects; such an actor is said to be *terminated*. Hence, garbage actors coincide with terminated actors in our model.

Terminated actors can be detected by looking at the global state of the system. We say that an actor B is a *potential acquaintance* of A (and A is a *potential inverse acquaintance* of B) if A has a reference to B or if there is an undelivered message to A that contains a reference to B. We define *potential reachability* to be the reflexive transitive closure of the potential acquaintance relation. If an actor is idle and has no undelivered messages, then it is *blocked*; otherwise it is *unblocked*. We then observe that an actor is terminated when it is only potentially reachable by blocked actors: Such an actor is idle, blocked, and can only potentially be sent a message by other idle blocked actors. Conversely, without analyzing actor code we cannot safely conclude that an actor is terminated if it is potentially reachable by an unblocked actor. Hence, we say that an actor is terminated if and only if it is blocked and all of its potential inverse acquaintances are terminated.

3 Related Work

Global Termination

Global termination detection (GTD) is used to determine when *all* processes have terminated [17, 16]. For GTD, it suffices to obtain global message send and receive counts. Most GTD algorithms also assume a fixed process topology. However, Lai gives an algorithm in [14] that supports dynamic topologies such as in the actor model. Lai's algorithm performs termination detection in "waves", disseminating control messages along a spanning tree (such as an actor supervisor hierarchy) so as to obtain consistent global message send and receive counts. Venkatasubramanian et al. take a similar approach to obtain a consistent global snapshot of actor states in a distributed system [25]. However, such an approach does not scale well because it is not incremental: garbage cannot be detected until all nodes in the system have responded. In contrast, DRL does not require a global snapshot, does not require actors to coordinate their local snapshots, and does not require waiting for all nodes before detecting local terminated actors.

Reference Tracking

We say that an idle actor is *simple garbage* if it has no undelivered messages and no other actor has a reference to it. Such actors can be detected with distributed reference counting [31, 6, 20] or with reference listing [21, 30] techniques. In reference listing algorithms, each actor maintains a partial list of actors that may have references to it. Whenever A sends B a reference to C, it also sends an `info` message informing C about B's reference. Once B no longer needs a reference to C, it informs C by sending a `release` message; this message should not be processed by C until all preceding messages from B to C have been delivered. Thus an actor is simple garbage when its reference listing is empty.

Our technique uses a form of *deferred reference listing*, in which A may also defer sending `info` messages to C until it releases its references to C. This allows `info` and `release` messages to be batched together, reducing communication overhead.

Cyclic Garbage

Actors that are transitively acquainted with one another are said to form cycles. Cycles of terminated actors are called *cyclic garbage* and cannot be detected with reference listing alone. Since actors are hosted on nodes and cycles may span across multiple nodes, detecting cyclic garbage requires sharing information between nodes to obtain a consistent view of the global topology. One approach is to compute a global snapshot of the distributed system [13] using the Chandy-Lamport algorithm [9]; but this requires pausing execution of all actors on a node to compute its local snapshot.

Another approach is to add edges to the actor reference graph so that actor garbage coincides with passive object garbage [24, 29]. This is convenient because it allows existing algorithms for distributed passive object GC, such as [23], to be reused in actor systems. However, such transformations require that actors know when they have undelivered messages, which requires some form of synchronization.

To avoid pausing executions, Wang and Varela proposed a reference listing based technique called the *pseudo-root* algorithm. The algorithm computes *approximate* global snapshots and is implemented in the SALSA runtime [30, 28]. The pseudo-root algorithm requires a high number of additional control messages and requires actors to write to shared memory if they migrate or release references during snapshot collection. Our protocol requires fewer control messages and no additional actions between local actor snapshots. Wang and Varela also explicitly address migration of actors, a concern orthogonal to our algorithm.

Our technique is inspired by *MAC*, a termination detection algorithm implemented in the Pony runtime [10]. In MAC, actors send a local snapshot to a designated cycle detector whenever their message queue becomes empty, and send another notification whenever it becomes non-empty. Clebsch and Drossopoulou prove that for systems with causal message delivery, a simple request-reply protocol is sufficient to confirm that the cycle detector's view of the topology is consistent. However, enforcing causal delivery in a distributed system imposes additional space and networking costs [11, 7]. DRL is similar to MAC, but does not require causal message delivery, supports decentralized termination detection, and actors need not take snapshots each time their message queues become empty. The key insight is that these limitations can be removed by tracking additional information at the actor level.

An earlier version of DRL appeared in [22]. In this paper, we formalize the description of the algorithm and prove its safety and liveness. In the process, we discovered that release acknowledgment messages are unnecessary and that termination detection is more flexible than we first thought: it is not necessary for GC to be performed in distinct "phases" where every actor takes a snapshot in each phase. In particular, once an idle actor takes a snapshot, it need not take another snapshot until it receives a fresh message.

4 A Two-Level Semantic Model

Our computation model is based on the two level approach to actor semantics [26], in which a lower *system-level* transition system interprets the operations performed by a higher, user-facing *application-level* transition system. In this section, we define the DRL communication protocol at the system level. We do not provide a transition system for the application level computation model, since it is not relevant to garbage collection (see [3] for how it can be done). What is relevant to us is that corresponding to each application-level action is a system-level transition that tracks references. We will therefore define *system-level configurations* and *transitions on system-level configurations*. We will refer to these, respectively, as configurations and transitions in the rest of the paper.

4.1 Overview

Actors in DRL use *reference objects* (abbreviated *refobs*) to send messages, instead of using plain actor addresses. Refobs are similar to unidirectional channels and can only be used by their designated *owner* to send messages to their *target*; thus in order for A to give B a reference to C, it must explicitly create a new refob owned by B. Once a refob is no longer needed, it should be *deactivated* by its owner and removed from local state.

The DRL communication protocol enriches each actor's state with a list of refobs that it currently owns and associated message counts representing the number of messages sent using each refob. Each actor also maintains a subset of the refobs of which it is the target, together with associated message receive counts. Lastly, actors perform a form of "contact tracing" by maintaining a subset of the refobs that they have created for other actors; we provide details about the bookkeeping later in this section.

The additional information above allows us to detect termination by inspecting actor snapshots. If a set of snapshots is consistent (in the sense of [9]) then we can use the "contact tracing" information to determine whether the set is *closed* under the potential inverse acquaintance relation (see Section 5). Then, given a consistent and closed set of snapshots, we can use the message counts to determine whether an actor is blocked. We can therefore find all the terminated actors within a consistent set of snapshots.

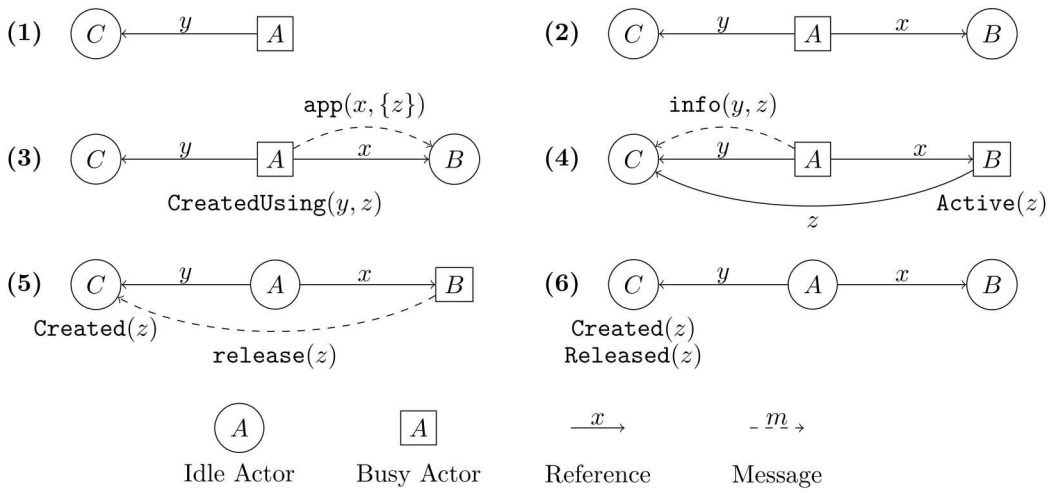

Figure 2 An example showing how refobs are created and destroyed. Below each actor we list all the "facts" related to z that are stored in its local state. Although not pictured in the figure, A also obtains facts $\texttt{Active}(x)$ and $\texttt{Active}(y)$ after spawning actors B and C, respectively. Likewise, actors B, C obtain facts $\texttt{Created}(x), \texttt{Created}(y)$, respectively, upon being spawned.

In fact, DRL satisfies a stronger property: any set of snapshots that "appears terminated" in the sense above is guaranteed to be consistent. Hence, given an arbitrary closed set of snapshots, it is possible to determine which of the corresponding actors have terminated. This allows a great deal of freedom in how snapshots are aggregated. For instance, actors could place their snapshots in a global eventually consistent store, with a garbage collection thread at each node periodically inspecting the store for local terminated actors.

Reference Objects

A refob is a triple (x, A, B), where A is the owner actor's address, B is the target actor's address, and x is a globally unique token. An actor can cheaply generate such a token by combining its address with a local sequence number, since actor systems already guarantee that each address is unique. We will stylize a triple (x, A, B) as $x : A \multimap B$. We will also sometimes refer to such a refob as simply x, since tokens act as unique identifiers.

When an actor A spawns an actor B (Fig. 2 (1, 2)) the DRL protocol creates a new refob $x : A \multimap B$ that is stored in both A and B's system-level state, and a refob $y : B \multimap B$ in B's state. The refob x allows A to send application-level messages to B. These messages are denoted $\texttt{app}(x, R)$, where R is the sett of refobs contained in the message that A has created for B. The refob y corresponds to the `self` variable present in some actor languages.

If A has active refobs $x : A \multimap B$ and $y : A \multimap C$, then it can create a new refob $z : B \multimap C$ by generating a token z. In addition to being sent to B, this refob must also temporarily be stored in A's system-level state and marked as "created using y" (Fig. 2 (3)). Once B receives z, it must add the refob to its system-level state and mark it as "active" (Fig. 2 (4)). Note that B can have multiple distinct refobs that reference the same actor in its state; this can be the result of, for example, several actors concurrently sending refobs to B. Transition rules for spawning actors and sending messages are given in Section 4.3.

Actor A may remove z from its state once it has sent a (system-level) `info` message informing C about z (Fig. 2 (4)). Similarly, when B no longer needs its refob for C, it can "deactivate" z by removing it from local state and sending C a (system-level) `release`

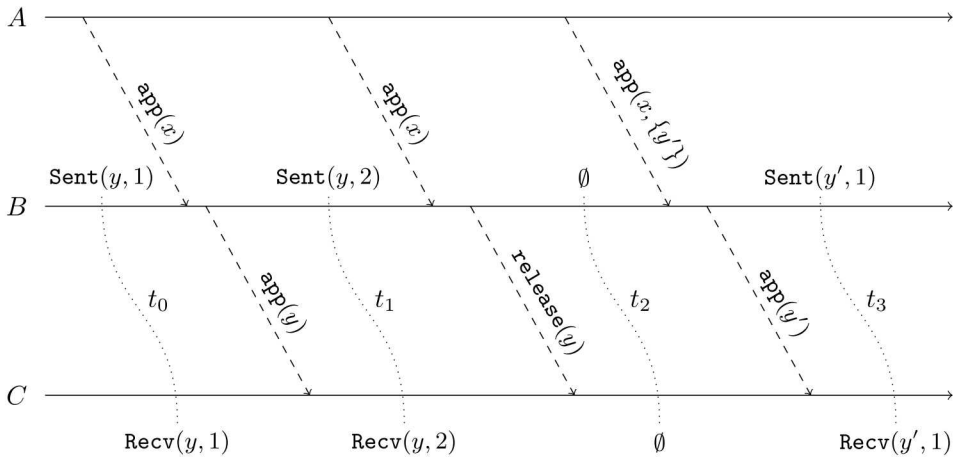

Figure 3 A time diagram for actors A, B, C, demonstrating message counts and consistent snapshots. Dashed arrows represent messages and dotted lines represent consistent cuts. In each cut above, B's message send count agrees with C's message receive count.

message (Fig. 2 (5)). Note that if B already has a refob $z : B \multimap C$ and then receives another $z' : B \multimap C$, then it can be more efficient to defer deactivating the extraneous z' until z is also no longer needed; this way, the `release` messages can be batched together.

When C receives an `info` message, it records that the refob has been created, and when C receives a `release` message, it records that the refob has been released (Fig. 2 (6)). Note that these messages may arrive in any order. Once C has received both, it is permitted to remove all facts about the refob from its local state. Transition rules for these reference listing actions are given in Section 4.4.

Once a refob has been created, it cycles through four states: pending, active, inactive, or released. A refob $z : B \multimap C$ is said to be *pending* until it is received by its owner B. Once received, the refob is *active* until it is *deactivated* by its owner, at which point it becomes *inactive*. Finally, once C learns that z has been deactivated, the refob is said to be *released*. A refob that has not yet been released is *unreleased*.

Slightly amending the definition we gave in Section 2, we say that B is a *potential acquaintance* of A (and A is a *potential inverse acquaintance* of B) when there exists an unreleased refob $x : A \multimap B$. Thus, B becomes a potential acquaintance of A as soon as x is created, and only ceases to be an acquaintance once it has received a `release` message for every refob $y : A \multimap B$ that has been created so far.

Message Counts and Snapshots

For each refob $x : A \multimap B$, the owner A counts the number of `app` and `info` messages sent along x; this count can be deleted when A deactivates x. Each message is annotated with the refob used to send it. Whenever B receives an `app` or `info` message along x, it correspondingly increments a receive count for x; this count can be deleted once x has been released. Thus the memory overhead of message counts is linear in the number of unreleased refobs.

A snapshot is a copy of all the facts in an actor's system-level state at some point in time. We will assume throughout the paper that in every set of snapshots Q, each snapshot was taken by a different actor. Such a set is also said to form a *cut*. Recall that a cut is consistent if no snapshot in the cut causally precedes any other [9]. Let us also say that Q is a set of *mutually quiescent* snapshots if there are no undelivered messages between actors

in the cut. That is, if $A \in Q$ sent a message to $B \in Q$ before taking a snapshot, then the message must have been delivered before B took its snapshot. Notice that if all snapshots in Q are mutually quiescent, then Q is consistent.

Notice also that in Fig. 3, the snapshots of B and C are mutually quiescent when their send and receive counts agree. This is ensured in part because each refob has a unique token: If actors associated message counts with actor names instead of tokens, then B's snapshots at t_0 and t_3 would both contain $\texttt{Sent}(C, 1)$. Thus, B's snapshot at t_3 and C's snapshot at t_0 would appear mutually quiescent, despite having undelivered messages in the cut.

We would like to conclude that snapshots from two actors A, B are mutually quiescent if and only if their send and receive counts are agreed for every refob $x : A \multimap B$ or $y : B \multimap A$. Unfortunately, this fails to hold in general for systems with unordered message delivery. It also fails to hold when, for instance, the owner actor takes a snapshot before the refob is activated and the target actor takes a snapshot after the refob is released. In such a case, neither knowledge set includes a message count for the refob and they therefore appear to agree. However, we show that the message counts can nevertheless be used to bound the number of undelivered messages for purposes of our algorithm (Lemma 12).

Definitions

We use the capital letters A, B, C, D, E to denote actor addresses. Tokens are denoted x, y, z, with a special reserved token `null` for messages from external actors.

A *fact* is a value that takes one of the following forms: $\texttt{Created}(x)$, $\texttt{Released}(x)$, $\texttt{CreatedUsing}(x, y)$, $\texttt{Active}(x)$, $\texttt{Unreleased}(x)$, $\texttt{Sent}(x, n)$, or $\texttt{Received}(x, n)$ for some refobs x, y and natural number n. Each actor's state holds a set of facts about refobs and message counts called its *knowledge set*. We use ϕ, ψ to denote facts and Φ, Ψ to denote finite sets of facts. Each fact may be interpreted as a *predicate* that indicates the occurrence of some past event. Interpreting a set of facts Φ as a set of axioms, we write $\Phi \vdash \phi$ when ϕ is derivable by first-order logic from Φ with the following additional rules:

- If $(\nexists n \in \mathbb{N},\ \texttt{Sent}(x, n) \in \Phi)$ then $\Phi \vdash \texttt{Sent}(x, 0)$
- If $(\nexists n \in \mathbb{N},\ \texttt{Received}(x, n) \in \Phi)$ then $\Phi \vdash \texttt{Received}(x, 0)$
- If $\Phi \vdash \texttt{Created}(x) \wedge \neg\texttt{Released}(x)$ then $\Phi \vdash \texttt{Unreleased}(x)$
- If $\Phi \vdash \texttt{CreatedUsing}(x, y)$ then $\Phi \vdash \texttt{Created}(y)$

For convenience, we define a pair of functions $\mathit{incSent}(x, \Phi)$, $\mathit{incRecv}(x, \Phi)$ for incrementing message send/receive counts, as follows: If $\texttt{Sent}(x, n) \in \Phi$ for some n, then $\mathit{incSent}(x, \Phi) = (\Phi \setminus \{\texttt{Sent}(x, n)\}) \cup \{\texttt{Sent}(x, n + 1)\}$; otherwise, $\mathit{incSent}(x, \Phi) = \Phi \cup \{\texttt{Sent}(x, 1)\}$. Likewise for *incRecv* and `Received`.

Recall that an actor is either *busy* (processing a message) or *idle* (waiting for a message). An actor with knowledge set Φ is denoted $[\Phi]$ if it is busy and (Φ) if it is idle.

Our specification includes both *system messages* (also called *control messages*) and *application messages*. The former are automatically generated by the DRL protocol and handled at the system level, whereas the latter are explicitly created and consumed by user-defined behaviors. Application-level messages are denoted $\texttt{app}(x, R)$. The argument x is the refob used to send the message. The second argument R is a set of refobs created by the sender to be used by the destination actor. Any remaining application-specific data in the message is omitted in our notation.

The DRL communication protocol uses two kinds of system messages. $\texttt{info}(y, z, B)$ is a message sent from an actor A to an actor C, informing it that a new refob $z : B \multimap C$ was created using $y : A \multimap C$. $\texttt{release}(x, n)$ is a message sent from an actor A to an actor B, informing it that the refob $x : A \multimap B$ has been deactivated and should be released.

A *configuration* $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$ is a quadruple $(\alpha, \mu, \rho, \chi)$ where: α is a mapping from actor addresses to knowledge sets; μ is a mapping from actor addresses to multisets of messages; and ρ, χ are sets of actor addresses. Actors in $\mathrm{dom}(\alpha)$ are *internal actors* and actors in χ are *external actors*; the two sets may not intersect. The mapping μ associates each actor with undelivered messages to that actor. Actors in ρ are *receptionists*. We will ensure $\rho \subseteq \mathrm{dom}(\alpha)$ remains valid in any configuration that is derived from a configuration where the property holds (referred to as the locality laws in [12]).

Configurations are denoted by κ, κ', κ_0, etc. If an actor address A (resp. a token x), does not occur in κ, then the address (resp. the token) is said to be *fresh*. We assume a facility for generating fresh addresses and tokens.

In order to express our transition rules in a pattern-matching style, we will employ the following shorthand. Let $\alpha, [\Phi]_A$ refer to a mapping α' where $\alpha'(A) = [\Phi]$ and $\alpha = \alpha'|_{\mathrm{dom}(\alpha')\setminus\{A\}}$. Similarly, let $\mu, [A \triangleleft m]$ refer to a mapping μ' where $m \in \mu'(A)$ and $\mu = \mu'|_{\mathrm{dom}(\mu')\setminus\{A\}} \cup \{A \mapsto \mu'(A) \setminus \{m\}\}$. Informally, the expression $\alpha, [\Phi]_A$ refers to a set of actors containing both α and the busy actor A (with knowledge set Φ); the expression $\mu, [A \triangleleft m]$ refers to the set of messages containing both μ and the message m (sent to actor A).

The rules of our transition system define atomic transitions from one configuration to another. Each transition rule has a label l, parameterized by some variables $\vec{x}$ that occur in the left- and right-hand configurations. Given a configuration κ, these parameters functionally determine the next configuration κ'. Given arguments $\vec{v}$, we write $\kappa \xrightarrow{l(\vec{v})} \kappa'$ to denote a semantic step from κ to κ' using rule $l(\vec{v})$.

We refer to a label with arguments $l(\vec{v})$ as an *event*, denoted e. A sequence of events is denoted π. If $\pi = e_1, \ldots, e_n$ then we write $\kappa \xrightarrow{\pi} \kappa'$ when $\kappa \xrightarrow{e_1} \kappa_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \kappa'$. If there exists π such that $\kappa \xrightarrow{\pi} \kappa'$, then κ' is *derivable* from κ. An *execution* is a sequence of events $e_1, \ldots, e_n$ such that $\kappa_0 \xrightarrow{e_1} \kappa_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \kappa_n$, where κ_0 is the initial configuration (Section 4.2). We say that a property holds *at time t* if it holds in κ_t.

4.2 Initial Configuration

The initial configuration κ_0 consists of a single actor in a busy state:

$$\langle\!\langle\ [\Phi]_A \mid \emptyset\ \rangle\!\rangle^{\emptyset}_{\{E\}},$$

where $\Phi = \{\texttt{Active}(x : A \multimap E),\ \texttt{Created}(y : A \multimap A),\ \texttt{Active}(y : A \multimap A)\}$. The actor's knowledge set includes a refob to itself and a refob to an external actor E. A can become a receptionist by sending E a refob to itself. Henceforth, we will only consider configurations that are derivable from an initial configuration.

4.3 Standard Actor Operations

Fig. 4 gives transition rules for standard actor operations, such as spawning actors and sending messages. Each of these rules corresponds a rule in the standard operational semantics of actors [3]. Note that each rule is atomic, but can just as well be implemented as a sequence of several smaller steps without loss of generality because actors do not share state – see [3] for a formal proof.

The Spawn event allows a busy actor A to spawn a new actor B and creates two refobs $x : A \multimap B,\ y : B \multimap B$. B is initialized with knowledge about x and y via the facts $\texttt{Created}(x), \texttt{Created}(y)$. The facts $\texttt{Active}(x), \texttt{Active}(y)$ allow A and B to immediately begin sending messages to B. Note that implementing Spawn does not require a synchronization protocol between A and B to construct $x : A \multimap B$. The parent A can pass both its

$\textsc{Spawn}(x, A, B)$

$$\langle\!\langle\ \alpha, [\Phi]_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, [\Phi \cup \{\texttt{Active}(x : A \multimap B)\}]_A, [\Psi]_B \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where x, y, B fresh
and $\Psi = \{\texttt{Created}(x : A \multimap B),\ \texttt{Created}(y : B \multimap B),\ \texttt{Active}(y : B \multimap B)\}$

$\textsc{Send}(x, \vec{y}, \vec{z}, A, B, \vec{C})$

$$\langle\!\langle\ \alpha, [\Phi]_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, [\mathit{incSent}(x, \Phi) \cup \Psi]_A \mid \mu, [B \triangleleft \texttt{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\vec{y}$ and $\vec{z}$ fresh and $n = |\vec{y}| = |\vec{z}| = |\vec{C}|$
and $\Phi \vdash \texttt{Active}(x : A \multimap B)$ and $\forall i \leq n,\ \Phi \vdash \texttt{Active}(y_i : A \multimap C_i)$
and $R = \{z_i : B \multimap C_i \mid i \leq n\}$ and $\Psi = \{\texttt{CreatedUsing}(y_i, z_i) \mid i \leq n\}$

$\textsc{Receive}(x, B, R)$

$$\langle\!\langle\ \alpha, (\Phi)_B \mid \mu, [B \triangleleft \texttt{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, [\mathit{incRecv}(x, \Phi) \cup \Psi]_B \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Active}(z) \mid z \in R\}$

$\textsc{Idle}(A)$

$$\langle\!\langle\ \alpha, [\Phi]_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, (\Phi)_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

Figure 4 Rules for standard actor interactions.

address and the freshly generated token x to the constructor for B. Since actors typically know their own addresses, this allows B to construct the triple (x, A, B). Since the `spawn` call typically returns the address of the spawned actor, A can also create the same triple.

The SEND event allows a busy actor A to send an application-level message to B containing a set of refobs $z_1, \ldots, z_n$ to actors $\vec{C} = C_1, \ldots, C_n$ – it is possible that $B = A$ or $C_i = A$ for some i. For each new refob z_i, we say that the message *contains* z_i. Any other data in the message besides these refobs is irrelevant to termination detection and therefore omitted. To send the message, A must have active refobs to both the target actor B and to every actor $C_1, \ldots, C_n$ referenced in the message. For each target C_i, A adds a fact $\texttt{CreatedUsing}(y_i, z_i)$ to its knowledge set; we say that A *created* z_i *using* y_i. Finally, A must increment its `Sent` count for the refob x used to send the message; we say that the message is sent *along* x.

The RECEIVE event allows an idle actor B to become busy by consuming an application message sent to B. Before performing subsequent actions, B increments the receive count for x and adds all refobs in the message to its knowledge set.

Finally, the IDLE event puts a busy actor into the idle state, enabling it to consume another message.

4.4 Release Protocol

Whenever an actor creates or receives a refob, it adds facts to its knowledge set. To remove these facts when they are no longer needed, actors can perform the *release protocol* defined in Fig. 5. All of these rules are not present in the standard operational semantics of actors.

SendInfo(y, z, A, B, C)

$$\langle\!\langle\ \alpha, [\Phi \cup \Psi]_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, [\mathit{incSent}(y, \Phi)]_A \mid \mu, [C \triangleleft \texttt{info}(y, z, B)]\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{CreatedUsing}(y : A \multimap C, z : B \multimap C)\}$

Info(y, z, B, C)

$$\langle\!\langle\ \alpha, (\Phi)_C \mid \mu, [C \triangleleft \texttt{info}(y, z, B)]\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, (\mathit{incRecv}(y, \Phi) \cup \Psi)_C \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Created}(z : B \multimap C)\}$

SendRelease(x, A, B)

$$\langle\!\langle\ \alpha, [\Phi \cup \Psi]_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, [\Phi]_A \mid \mu, [B \triangleleft \texttt{release}(x, n)]\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Active}(x : A \multimap B), \texttt{Sent}(x, n)\}$
and $\nexists y,\ \texttt{CreatedUsing}(x, y) \in \Phi$

Release(x, A, B)

$$\langle\!\langle\ \alpha, (\Phi)_B \mid \mu, [B \triangleleft \texttt{release}(x, n)]\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, (\Phi \cup \{\texttt{Released}(x)\})_B \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

only if $\Phi \vdash \texttt{Received}(x, n)$

Compaction(x, B, C)

$$\langle\!\langle\ \alpha, (\Phi \cup \Psi)_C \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, (\Phi)_C \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

where $\Psi = \{\texttt{Created}(x : B \multimap C), \texttt{Released}(x : B \multimap C), \texttt{Received}(x, n)\}$ for some $n \in \mathbb{N}$
or $\Psi = \{\texttt{Created}(x : B \multimap C), \texttt{Released}(x : B \multimap C)\}$ and $\forall n \in \mathbb{N},\ \texttt{Received}(x, n) \notin \Phi$

Snapshot(A, Φ)

$$\langle\!\langle\ \alpha, (\Phi)_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha, (\Phi)_A \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

Figure 5 Rules for performing the release protocol.

The SendInfo event allows a busy actor A to inform C about a refob $z : B \multimap C$ that it created using y; we say that the `info` message is sent *along* y and *contains* z. This event allows A to remove the fact $\texttt{CreatedUsing}(y, z)$ from its knowledge set. It is crucial that A also increments its `Sent` count for y to indicate an undelivered `info` message sent to C: it allows the snapshot aggregator to detect when there are undelivered `info` messages, which contain refobs. This message is delivered with the Info event, which adds the fact $\texttt{Created}(z : B \multimap C)$ to C's knowledge set and correspondingly increments C's `Received` count for y.

When an actor A no longer needs $x : A \multimap B$ for sending messages, A can deactivate x with the SendRelease event; we say that the `release` is sent *along* x. A precondition of this event is that A has already sent messages to inform B about all the refobs it has created using x. In practice, an implementation may defer sending any `info` or `release` messages to a target B until all A's refobs to B are deactivated. This introduces a trade-off between the number of control messages and the rate of simple garbage detection (Section 5).

$\textsc{In}(A, R)$

$$\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha \mid \mu, [A \triangleleft \texttt{app}(\texttt{null}, R)]\ \rangle\!\rangle^{\rho}_{\chi \cup \chi'}$$

where $A \in \rho$ and $R = \{x_1 : A \multimap B_1, \ldots, x_n : A \multimap B_n\}$ and $x_1, \ldots, x_n$ fresh
and $\{B_1, \ldots, B_n\} \cap \mathrm{dom}(\alpha) \subseteq \rho$ and $\chi' = \{B_1, \ldots, B_n\} \setminus \mathrm{dom}(\alpha)$

$\textsc{Out}(x, B, R)$

$$\langle\!\langle\ \alpha \mid \mu, [B \triangleleft \texttt{app}(x, R)]\ \rangle\!\rangle^{\rho}_{\chi} \to \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho \cup \rho'}_{\chi}$$

where $B \in \chi$ and $R = \{x_1 : B \multimap C_1, \ldots, x_n : B \multimap C_n\}$ and $\rho' = \{C_1, \ldots, C_n\} \cap \mathrm{dom}(\alpha)$

$\textsc{ReleaseOut}(x, B)$

$$\langle\!\langle\ \alpha \mid \mu, [B \triangleleft \texttt{release}(x, n)]\ \rangle\!\rangle^{\rho}_{\chi \cup \{B\}} \to \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi \cup \{B\}}$$

$\textsc{InfoOut}(y, z, A, B, C)$

$$\langle\!\langle\ \alpha \mid \mu, [C \triangleleft \texttt{info}(y, z, A, B)]\ \rangle\!\rangle^{\rho}_{\chi \cup \{C\}} \to \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi \cup \{C\}}$$

Figure 6 Rules for interacting with the outside world.

Each `release` message for a refob x includes a count n of the number of messages sent using x. This ensures that $\texttt{release}(x, n)$ is only delivered after all the preceding messages sent along x have been delivered. Once the RELEASE event can be executed, it adds the fact that x has been released to B's knowledge set. Once C has received both an `info` and `release` message for a refob x, it may remove facts about x from its knowledge set using the COMPACTION event.

Finally, the SNAPSHOT event captures an idle actor's knowledge set. For simplicity, we have omitted the process of disseminating snapshots to an aggregator. Although this event does not change the configuration, it allows us to prove properties about snapshot events at different points in time.

4.5 Composition and Effects

We give rules to dictate how internal actors interact with external actors in Fig. 6. The IN and OUT rules correspond to similar rules in the standard operational semantics of actors.

Since internal garbage collection protocols are not exposed to the outside world, all `release` and `info` messages sent to external actors are simply dropped by the RELEASEOUT and INFOOUT events. Likewise, only `app` messages can enter the system. Since we cannot statically determine when a receptionist's address has been forgotten by all external actors, we assume that receptionists are never terminated. The resulting "black box" behavior of our system is the same as the actor systems in [3]. Hence, in principle DRL can be gradually integrated into a codebase by creating a subsystem for garbage-collected actors.

The IN event allows an external actor to send an application-level message to a receptionist A containing a set of refobs R, all owned by A. Since external actors do not use refobs, the message is sent using the special `null` token. All targets in R that are not internal actors are added to the set of external actors.

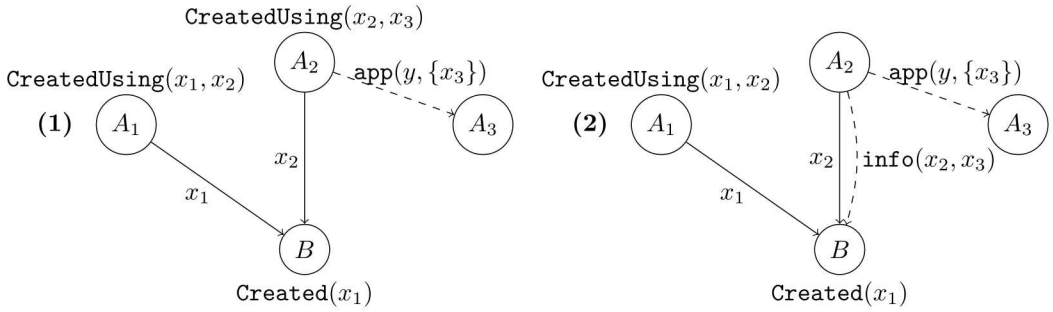

Figure 7 An example of a chain from B to x_3.

The Out event delivers an application-level message to an external actor with a set of refobs R. All internal actors referenced in R become receptionists because their addresses have been exposed to the outside world.

4.6 Garbage

We can now operationally characterize actor garbage in our model. An actor A can *potentially receive a message* in κ if there is a sequence of events (possibly of length zero) leading from κ to a configuration κ' in which A has an undelivered message. We say that an actor is *terminated* if it is idle and cannot potentially receive a message.

An actor is *blocked* if it satisfies three conditions: (1) it is idle, (2) it is not a receptionist, and (3) it has no undelivered messages; otherwise, it is *unblocked*. We define *potential reachability* as the reflexive transitive closure of the potential acquaintance relation. That is, A_1 can potentially reach A_n if and only if there is a sequence of unreleased refobs $(x_1 : A_1 \multimap A_2), \ldots, (x_n : A_{n-1} \multimap A_n)$; recall that a refob $x : A \multimap B$ is unreleased if its target B has not yet received a `release` message for x.

Notice that an actor can potentially receive a message if and only if it is potentially reachable from an unblocked actor. Hence an actor is terminated if and only if it is only potentially reachable by blocked actors. A special case of this is *simple garbage*, in which an actor is blocked and has no potential inverse acquaintances besides itself.

We say that a set of actors S is *closed* (with respect to the potential inverse acquaintance relation) if, whenever $B \in S$ and there is an unreleased refob $x : A \multimap B$, then also $A \in S$. Notice that the closure of a set of terminated actors is also a set of terminated actors.

5 Chain Lemma

To determine if an actor has terminated, one must show that all of its potential inverse acquaintances have terminated. This appears to pose a problem for termination detection, since actors cannot have a complete listing of all their potential inverse acquaintances without some synchronization: actors would need to consult their acquaintances before creating new references to them. In this section, we show that the DRL protocol provides a weaker guarantee that will nevertheless prove sufficient: knowledge about an actor's refobs is *distributed* across the system and there is always a "path" from the actor to any of its potential inverse acquaintances.

Let us construct a concrete example of such a path, depicted by Fig. 7. Suppose that A_1 spawns B, gaining a refob $x_1 : A_1 \multimap B$. Then A_1 may use x_1 to create $x_2 : A_2 \multimap B$, which A_2 may receive and then use x_2 to create $x_3 : A_3 \multimap B$.

At this point, there are unreleased refobs owned by A_2 and A_3 that are not included in B's knowledge set. However, Fig. 7 shows that the distributed knowledge of B, A_1, A_2 creates a "path" to all of B's potential inverse acquaintances. Since A_1 spawned B, B knows the fact $\texttt{Created}(x_1)$. Then when A_1 created x_2, it added the fact $\texttt{CreatedUsing}(x_1, x_2)$ to its knowledge set, and likewise A_2 added the fact $\texttt{CreatedUsing}(x_2, x_3)$; each fact points to another actor that owns an unreleased refob to B (Fig. 7 (1)).

Since actors can remove `CreatedUsing` facts by sending `info` messages, we also consider (Fig. 7 (2)) to be a "path" from B to A_3. But notice that, once B receives the `info` message, the fact $\texttt{Created}(x_3)$ will be added to its knowledge set and so there will be a "direct path" from B to A_3. We formalize this intuition with the notion of a *chain* in a given configuration $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$:

▶ **Definition 1.** *A* chain to $x : A \multimap B$ *is a sequence of unreleased refobs* $(x_1 : A_1 \multimap B), \ldots,$ $(x_n : A_n \multimap B)$ *such that:*

- $\alpha(B) \vdash \texttt{Created}(x_1 : A_1 \multimap B)$*;*
- *For all* $i < n$*, either* $\alpha(A_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$ *or the message* $[B \triangleleft \texttt{info}(x_i, x_{i+1})]$ *is in transit; and*
- $A_n = A$ *and* $x_n = x$*.*

We say that an actor B is *in the root set* if it is a receptionist or if there is an application message $\texttt{app}(x, R)$ in transit to an external actor with $B \in \text{targets}(R)$. Since external actors never release refobs, actors in the root set must never terminate.

▶ **Lemma 2** (Chain Lemma)**.** *Let* B *be an internal actor in* κ*. If* B *is not in the root set, then there is a chain to every unreleased refob* $x : A \multimap B$*. Otherwise, there is a chain to some refob* $y : C \multimap B$ *where* C *is an external actor.*

▶ Remark. When B is in the root set, not all of its unreleased refobs are guaranteed to have chains. This is because an external actor may send B's address to other receptionists without sending an `info` message to B.

An immediate application of the Chain Lemma is to allow actors to detect when they are simple garbage. If any actor besides B owns an unreleased refob to B, then B must have a fact $\texttt{Created}(x : A \multimap B)$ in its knowledge set where $A \neq B$. Hence, if B has no such facts, then it must have no nontrivial potential inverse acquaintances. Moreover, since actors can only have undelivered messages along unreleased refobs, B also has no undelivered messages from any other actor; it can only have undelivered messages that it sent to itself. This gives us the following result:

▶ **Theorem 3.** *Suppose* B *is idle with knowledge set* Φ*, such that:*

- Φ *does not contain any facts of the form* $\texttt{Created}(x : A \multimap B)$ *where* $A \neq B$*; and*
- *for all facts* $\texttt{Created}(x : B \multimap B) \in \Phi$*, also* $\Phi \vdash \texttt{Sent}(x, n) \wedge \texttt{Received}(x, n)$ *for some* n*.*

Then B *is simple garbage.*

6 Termination Detection

In order to detect non-simple terminated garbage, actors periodically sends a snapshot of their knowledge set to a snapshot aggregator actor. An aggregator in turn may disseminate snapshots it has to other aggregators. Each aggregator maintains a map data structure, associating an actor's address to its most recent snapshot; in effect, snapshot aggregators

maintain an eventually consistent key-value store with addresses as keys and snapshots as values. At any time, an aggregator can scan its local store to find terminated actors and send them a request to self-destruct.

Given an arbitrary set of snapshots Q, we characterize the *finalized subsets* of Q in this section. We show that the actors that took these finalized snapshots must be terminated. Conversely, the snapshots of any closed set of terminated actors are guaranteed to be finalized. (Recall that the closure of a set of terminated actors is also a terminated set of actors.) Thus, snapshot aggregators can eventually detect all terminated actors by periodically searching their local stores for finalized subsets. Finally, we give an algorithm for obtaining the maximum finalized subset of a set Q by "pruning away" the snapshots of actors that appear not to have terminated.

Recall that when we speak of a set of snapshots Q, we assume each snapshot was taken by a different actor. We will write $\Phi_A \in Q$ to denote A's snapshot in Q; we will also write $A \in Q$ if A has a snapshot in Q. We will also write $Q \vdash \phi$ if $\Phi \vdash \phi$ for some $\Phi \in Q$.

▶ **Definition 4.** *A set of snapshots Q is* closed *if, whenever $Q \vdash \texttt{Unreleased}(x : A \multimap B)$ and $B \in Q$, then also $A \in Q$ and $\Phi_A \vdash \texttt{Active}(x : A \multimap B)$.*

▶ **Definition 5.** *An actor $B \in Q$* appears blocked *if, for every $Q \vdash \texttt{Unreleased}(x : A \multimap B)$, then $\Phi_A, \Phi_B \in Q$ and $\Phi_A \vdash \texttt{Sent}(x, n)$ and $\Phi_B \vdash \texttt{Received}(x, n)$ for some n.*

▶ **Definition 6.** *A set of snapshots Q is* finalized *if it is closed and every actor in Q appears blocked.*

This definition corresponds to our characterization in Section 4.6: An actor is terminated precisely when it is in a closed set of blocked actors.

▶ **Theorem 7** (Safety)**.** *If Q is a finalized set of snapshots at time t_f then the actors in Q are all terminated at t_f.*

We say that the *final action* of a terminated actor is the last non-snapshot event it performs before becoming terminated. Notice that an actor's final action can only be an Idle, Info, or Release event. Note also that the final action may come *strictly before* an actor becomes terminated, since a blocked actor may only terminate after all of its potential inverse acquaintances become blocked.

The following lemma allows us to prove that DRL is eventually live. It also shows that an non-finalized set of snapshots must have an unblocked actor.

▶ **Lemma 8.** *Let S be a closed set of terminated actors at time t_f. If every actor in S took a snapshot sometime after its final action, then the resulting set of snapshots is finalized.*

▶ **Theorem 9** (Liveness)**.** *If every actor eventually takes a snapshot after performing an Idle, Info, or Release event, then every terminated actor is eventually part of a finalized set of snapshots.*

Proof. If an actor A is terminated, then the closure S of $\{A\}$ is a terminated set of actors. Since every actor eventually takes a snapshot after taking its final action, Lemma 8 implies that the resulting snapshots of S are finalized. ◀

We say that a refob $x : A \multimap B$ is *unreleased* in Q if $Q \vdash \texttt{Unreleased}(x)$. Such a refob is said to be *relevant* when $B \in Q$ implies $A \in Q$ and $\Phi_A \vdash \texttt{Active}(x)$ and $\Phi_A \vdash \texttt{Sent}(x, n)$ and $\Phi_B \vdash \texttt{Received}(x, n)$ for some n; intuitively, this indicates that B has no undelivered messages along x. Notice that a set Q is finalized if and only if all unreleased refobs in Q are relevant.

Observe that if $x : A \multimap B$ is unreleased and irrelevant in Q, then B cannot be in any finalized subset of Q. We can therefore employ a simple iterative algorithm to find the maximum finalized subset of Q: for each irrelevant unreleased refob $x : A \multimap B$ in Q, remove the target B from Q. Since this can make another unreleased refob $y : B \multimap C$ irrelevant, we must repeat this process until a fixed point is reached. In the resulting subset Q', all unreleased refobs are relevant. Since all actors in $Q \setminus Q'$ are not members of any finalized subset of Q, it must be that Q' is the maximum finalized subset of Q.

7 Conclusion and Future Work

We have shown how deferred reference listing and message counts can be used to detect termination in actor systems. The technique is provably safe (Theorem 7) and eventually live (Theorem 9). An implementation in Akka is presently underway.

We believe that DRL satisfies our three initial goals:

1. *Termination detection does not restrict concurrency in the application.* Actors do not need to coordinate their snapshots or pause execution during garbage collection.
2. *Termination detection does not impose high overhead.* The amortized memory overhead of our technique is linear in the number of unreleased refobs. Besides application messages, the only additional control messages required by the DRL communication protocol are `info` and `release` messages. These control messages can be batched together and deferred, at the cost of worse termination detection time.
3. *Termination detection scales with the number of nodes in the system.* Our algorithm is incremental, decentralized, and does not require synchronization between nodes.

Since it does not matter what order snapshots are collected in, DRL can be used as a "building block" for more sophisticated garbage collection algorithms. One promising direction is to take a *generational* approach [15], in which long-lived actors take snapshots less frequently than short-lived actors. Different types of actors could also take snapshots at different rates. In another approach, snapshot aggregators could *request* snapshots instead of waiting to receive them.

In the presence of faults, DRL remains safe but its liveness properties are affected. If an actor A crashes and its state cannot be recovered, then none of its refobs can be released and the aggregator will never receive its snapshot. Consequently, all actors potentially reachable from A can no longer be garbage collected. However, A's failure does not affect the garbage collection of actors it cannot reach. In particular, network partitions between nodes will not delay node-local garbage collection.

Choosing an adequate fault-recovery protocol will likely vary depending on the target actor framework. One option is to use checkpointing or event-sourcing to persist GC state; the resulting overhead may be acceptable in applications that do not frequently spawn actors or create refobs. Another option is to monitor actors for failure and infer which refobs are no longer active; this is a subject for future work.

Another issue that can affect liveness is message loss: If any messages along a refob $x : A \multimap B$ are dropped, then B can never be garbage collected because it will always appear unblocked. This is, in fact, the desired behavior if one cannot guarantee that the message will not be delivered at some later point. In practice, this problem might be addressed with watermarking.

References

1 Gul Agha. *ACTORS – a Model of Concurrent Computation in Distributed Systems*. MIT Press Series in Artificial Intelligence. MIT Press, 1990.

2 Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990. doi:10.1145/83880.84528.

3 Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, January 1997. doi:10.1017/S095679689700261X.

4 Akka. URL: https://akka.io/.

5 Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1996.

6 Di Bevan. Distributed garbage collection using reference counting. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, J. W. Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, volume 259, pages 176–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987. doi:10.1007/3-540-17945-3_10.

7 Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. Tree topologies for causal message delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2017*, pages 1–10, Vancouver, BC, Canada, 2017. ACM Press. doi:10.1145/3141834.3141835.

8 Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, pages 1–14, Cascais, Portugal, 2011. ACM Press. doi:10.1145/2038916.2038932.

9 K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985. doi:10.1145/214451.214456.

10 Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '13*, pages 553–570, Indianapolis, Indiana, USA, 2013. ACM Press. doi:10.1145/2509136.2509557.

11 Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.

12 Carl Hewitt and Henry G. Baker. Laws for communicating parallel processes. In Bruce Gilchrist, editor, *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, pages 987–992. North-Holland, 1977.

13 D. Kafura, M. Mukherji, and D.M. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):337–350, April 1995. doi:10.1109/71.372788.

14 Ten-Hwang Lai. Termination detection for dynamically distributed systems with non-first-in-first-out communication. *Journal of Parallel and Distributed Computing*, 3(4):577–599, December 1986. doi:10.1016/0743-7315(86)90015-8.

15 Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983. doi:10.1145/358141.358147.

16 Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, November 1998. doi:10.1016/S0164-1212(98)10034-1.

17 Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, September 1987. doi:10.1007/BF01782776.

18 NHS to Deploy Riak for New IT Backbone With Quality of Care Improvements in Sight, October 2013. https://riak.com/nhs-to-deploy-riak-for-new-it-backbone-with-quality-of-care-improvements-in-sight.html.

19 PayPal Blows Past 1 Billion Transactions Per Day Using Just 8 VMs With Akka, Scala, Kafka and Akka Streams. https://www.lightbend.com/case-studies/paypal-blows-past-1-billion-transactions-per-day-using-just-8-vms-and-akka-scala-kafka-and-akka-streams.

20 José M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem, editors, *Parle '91 Parallel Architectures and Languages Europe*, volume 505, pages 150–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. doi:10.1007/978-3-662-25209-3_11.

21 David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings*, pages 211–249, 1995. doi:10.1007/3-540-60368-9_26.

22 Dan Plyukhin and Gul Agha. Concurrent garbage collection in the actor model. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2018*, pages 44–53, Boston, MA, USA, 2018. ACM Press. doi:10.1145/3281366.3281368.

23 M. Schelvis. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications - OOPSLA '89*, pages 37–48, New Orleans, Louisiana, United States, 1989. ACM Press. doi:10.1145/74877.74883.

24 Abhay Vardhan and Gul Agha. Using passive object garbage collection algorithms for garbage collection of active objects. *ACM SIGPLAN Notices*, 38(2 supplement):106, February 2003. doi:10.1145/773039.512443.

25 Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637, pages 134–147. Springer-Verlag, Berlin/Heidelberg, 1992. doi:10.1007/BFb0017187.

26 Nalini Venkatasubramanian and Carolyn Talcott. Reasoning about meta level activities in open distributed systems. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing - PODC '95*, pages 144–152, Ottowa, Ontario, Canada, 1995. ACM Press. doi:10.1145/224964.224981.

27 Stanislav Vishnevskiy. How Discord Scaled Elixir to 5,000,000 Concurrent Users, July 2017. URL: https://blog.discord.com/scaling-elixir-f9b8e1e7c29b.

28 Wei-Jen Wang. Conservative snapshot-based actor garbage collection for distributed mobile actor systems. *Telecommunication Systems*, June 2011. doi:10.1007/s11235-011-9509-1.

29 Wei-Jen Wang, Carlos Varela, Fu-Hau Hsu, and Cheng-Hsien Tang. Actor Garbage Collection Using Vertex-Preserving Actor-to-Object Graph Transformations. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, and Peter M. A. Sloot, editors, *Advances in Grid and Pervasive Computing*, volume 6104, pages 244–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-13067-0_28.

30 Wei-Jen Wang and Carlos A. Varela. Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach. In Yeh-Ching Chung and José E. Moreira, editors, *Advances in Grid and Pervasive Computing*, volume 3947, pages 360–372. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/11745693_36.

31 Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, J. W. Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, volume 259, pages 432–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987. doi:10.1007/3-540-17945-3_25.

A Appendix

A.1 Basic Properties

▶ **Lemma 10.** *If B has undelivered messages along $x : A \multimap B$, then x is an unreleased refob.*

Proof. There are three types of messages: `app`, `info`, and `release`. All three messages can only be sent when x is active. Moreover, the RELEASE rule ensures that they must all be delivered before x can be released. ◀

▶ **Lemma 11.**

- *Once* $\texttt{CreatedUsing}(y : A \multimap C, z : B \multimap C)$ *is added to A's knowledge set, it will not be removed until after A has sent an* `info` *message containing z to C.*
- *Once* $\texttt{Created}(z : B \multimap C)$ *is added to C's knowledge set, it will not be removed until after C has received the (unique)* `release` *message along z.*
- *Once* $\texttt{Released}(z : B \multimap C)$ *is added to C's knowledge set, it will not be removed until after C has received the (unique)* `info` *message containing z.*

Proof. Immediate from the transition rules. ◀

▶ **Lemma 12.** *Consider a refob $x : A \multimap B$. Let t_1, t_2 be times such that x has not yet been deactivated at t_1 and x has not yet been released at t_2. In particular, t_1 and t_2 may be before the creation time of x.*

Suppose that $\alpha_{t_1}(A) \vdash \texttt{Sent}(x, n)$ and $\alpha_{t_2}(B) \vdash \texttt{Received}(x, m)$ and, if $t_1 < t_2$, that A does not send any messages along x during the interval $[t_1, t_2]$. Then the difference $\max(n - m, 0)$ is the number of messages sent along x before t_1 that were not received before t_2.

Proof. Since x is not deactivated at time t_1 and unreleased at time t_2, the message counts were never reset by the SENDRELEASE or COMPACTION rules. Hence n is the number of messages A sent along x before t_1 and m is the number of messages B received along x before t_2. Hence $\max(n - m, 0)$ is the number of messages sent before t_1 and *not* received before t_2. ◀

A.2 Chain Lemma

▶ **Lemma 2** (Chain Lemma). *Let B be an internal actor in κ. If B is not in the root set, then there is a chain to every unreleased refob $x : A \multimap B$. Otherwise, there is a chain to some refob $y : C \multimap B$ where C is an external actor.*

Proof. We prove that the invariant holds in the initial configuration and at all subsequent times by induction on events $\kappa \xrightarrow{e} \kappa'$, omitting events that do not affect chains. Let $\kappa = \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$ and $\kappa' = \langle\!\langle\ \alpha' \mid \mu'\ \rangle\!\rangle^{\rho'}_{\chi'}$.

In the initial configuration, the only refob to an internal actor is $y : A \multimap A$. Since A knows $\texttt{Created}(y : A \multimap A)$, the invariant is satisfied.

In the cases below, let x, y, z, A, B, C be free variables, not referencing the variables used in the statement of the lemma.

- SPAWN(x, A, B) creates a new unreleased refob $x : A \multimap B$, which satisfies the invariant because $\alpha'(B) \vdash \texttt{Created}(x : A \multimap B)$.

- $\textsc{Send}(x, \vec{y}, \vec{z}, A, B, \vec{C})$ creates a set of refobs R. Let $(z : B \multimap C) \in R$, created using $y : A \multimap C$.
  If C is already in the root set, then the invariant is trivially preserved. Otherwise, there must be a chain $(x_1 : A_1 \multimap C), \ldots, (x_n : A_n \multimap C)$ where $x_n = y$ and $A_n = A$. Then $x_1, \ldots, x_n, z$ is a chain in κ', since $\alpha'(A_n) \vdash \texttt{CreatedUsing}(x_n, z)$.
  If B is an internal actor, then this shows that every unreleased refob to C has a chain in κ'. Otherwise, C is in the root set in κ'. To see that the invariant still holds, notice that $z : B \multimap C$ is a witness of the desired chain.
- $\textsc{SendInfo}(y, z, A, B, C)$ removes the $\texttt{CreatedUsing}(y, z)$ fact but also sends $\texttt{info}(y, z, B)$, so chains are unaffected.
- $\textsc{Info}(y, z, B, C)$ delivers $\texttt{info}(y, z, B)$ to C and adds $\texttt{Created}(z : B \multimap C)$ to its knowledge set.
  Suppose $z : B \multimap C$ is part of a chain $(x_1 : A_1 \multimap C), \ldots, (x_n : A_n \multimap C)$, i.e. $x_i = y$ and $x_{i+1} = z$ and $A_{i+1} = B$ for some $i < n$. Since $\alpha'(C) \vdash \texttt{Created}(x_{i+1} : A_{i+1} \multimap C)$, we still have a chain $x_{i+1}, \ldots, x_n$ in κ'.
- $\textsc{Release}(x, A, B)$ releases the refob $x : A \multimap B$. Since external actors never release their refobs, both A and B must be internal actors.
  Suppose the released refob was part of a chain $(x_1 : A_1 \multimap B), \ldots, (x_n : A_n \multimap B)$, i.e. $x_i = x$ and $A_i = A$ for some $i < n$. We will show that $x_{i+1}, \ldots, x_n$ is a chain in κ'.
  Before performing $\textsc{SendRelease}(x_i, A_i, B)$, A_i must have performed the $\textsc{Info}(x_i, x_{i+1}, A_{i+1}, B)$ event. Since the $\texttt{info}$ message was sent along x_i, Lemma 10 ensures that the message must have been delivered before the present $\textsc{Release}$ event. Furthermore, since x_{i+1} is an unreleased refob in κ', Lemma 11 ensures that $\alpha'(B) \vdash \texttt{Created}(x_{i+1} : A_{i+1} \multimap B)$.
- $\textsc{In}(A, R)$ adds a message from an external actor to the internal actor A. This event can only create new refobs that point to receptionists, so it preserves the invariant.
- $\textsc{Out}(x, B, R)$ emits a message $\texttt{app}(x, R)$ to the external actor B. Since all targets in R are already in the root set, the invariant is preserved. ◀

A.3 Termination Detection

Given a set of snapshots Q taken before some time t_f, we write Q_t to denote those snapshots in Q that were taken before time $t < t_f$. If $\Phi_A \in Q$, we denote the time of A's snapshot as t_A.

▶ **Lemma 8.** *Let S be a closed set of terminated actors at time t_f. If every actor in S took a snapshot sometime after its final action, then the resulting set of snapshots is finalized.*

Call this set of snapshots Q. First, we prove the following lemma.

▶ **Lemma 13.** *If $Q \vdash \texttt{Unreleased}(x : A \multimap B)$ and $B \in Q$, then x is unreleased at t_B.*

Proof. By definition, $Q \vdash \texttt{Unreleased}(x : A \multimap B)$ only if $Q \vdash \texttt{Created}(x) \wedge \neg\texttt{Released}(x)$. Since $Q \nvdash \texttt{Released}(x)$, we must also have $\Phi_B \nvdash \texttt{Released}(x)$. For $Q \vdash \texttt{Created}(x)$, there are two cases.

Case 1: $\Phi_B \vdash \texttt{Created}(x)$. Since $\Phi_B \nvdash \texttt{Released}(x)$, Lemma 11 implies that x is unreleased at time t_B.

Case 2: For some $C \in Q$ and some y, $\Phi_C \vdash \texttt{CreatedUsing}(y, x)$. Since C performed its final action before taking its snapshot, this implies that C will never send the $\texttt{info}$ message containing x to B.

Suppose then for a contradiction that x is released at time t_B. Since $\Phi_B \not\vdash \texttt{Released}(x)$, Lemma 11 implies that B received an `info` message containing x before its snapshot. But this is impossible because C never sends this message. ◀

Proof (Lemma 8). By strong induction on time t, we show that Q is closed and that every actor appears blocked.

Induction hypothesis: For all times $t' < t$, if $B \in Q_{t'}$ and $Q \vdash \texttt{Unreleased}(x : A \multimap B)$, then $A \in Q$, $Q \vdash \texttt{Active}(x)$, and $Q \vdash \texttt{Sent}(x, n)$ and $Q \vdash \texttt{Received}(x, n)$ for some n.

Since $Q_0 = \emptyset$, the induction hypothesis holds trivially in the initial configuration.

Now assume the induction hypothesis. Suppose that $B \in Q$ takes its snapshot at time t with $Q \vdash \texttt{Unreleased}(x : A \multimap B)$, which implies $Q \vdash \texttt{Created}(x) \wedge \neg\texttt{Released}(x)$.

$Q \vdash \texttt{Created}(x)$ implies that x was created before t_f. Lemma 13 implies that x is also unreleased at time t_f, since B cannot perform a RELEASE event after its final action. Hence A is in the closure of $\{B\}$ at time t_f, so $A \in Q$.

Now suppose $\Phi_A \not\vdash \texttt{Active}(x)$. Then either x will be activated after t_A or x was deactivated before t_A. The former is impossible because A would need to become unblocked to receive x. Since x is unreleased at time t_f and $t_A < t_f$, the latter implies that there is an undelivered `release` message for x at time t_f. But this is impossible as well, since B is blocked at t_f.

Finally, let n such that $\Phi_B \vdash \texttt{Received}(x, n)$; we must show that $\Phi_A \vdash \texttt{Sent}(x, n)$. By the above arguments, x is active at time t_A and unreleased at time t_B. Since both actors performed their final action before their snapshots, all messages sent before t_A must have been delivered before t_B. By Lemma 12, this implies $\Phi_A \vdash \texttt{Sent}(x, n)$. ◀

We now prove the safety theorem, which states that if Q is a finalized set of snapshots, then the corresponding actors of Q are terminated. We do this by showing that at each time t, all actors in Q_t are blocked and all of their potential inverse acquaintances are in Q.

Consider the first actor B in Q to take a snapshot. We show, using the Chain Lemma, that the closure of this actor is in Q. Then, since all potential inverse acquaintances of B take snapshots strictly after t_B, it is impossible for B to have any undelivered messages without appearing unblocked.

For every subsequent actor B to take a snapshot, we make a similar argument with an additional step: If B has any potential inverse acquaintances in Q_{t_B}, then they could not have sent B a message without first becoming unblocked.

▶ **Theorem 7** (Safety). *If Q is a finalized set of snapshots at time t_f then the actors in Q are all terminated at t_f.*

Proof. Proof by induction on events. The induction hypothesis consists of two clauses that must both be satisfied at all times $t \leq t_f$.

- **IH 1:** If $B \in Q_t$ and $x : A \multimap B$ is unreleased, then $Q \vdash \texttt{Unreleased}(x)$.
- **IH 2:** The actors of Q_t are all blocked.

Initial configuration. Since $Q_0 = \emptyset$, the invariant trivially holds.

Snapshot(B, Φ_B). Suppose $B \in Q$ takes a snapshot at time t. We show that if $x : A \multimap B$ is unreleased at time t, then $Q \vdash \texttt{Unreleased}(x)$ and there are no undelivered messages along x from A to B. We do this with the help of two lemmas.

▶ **Lemma 14.** *If $Q \vdash \texttt{Unreleased}(x : A \multimap B)$, then x is unreleased at time t and there are no undelivered messages along x at time t. Moreover, if $t_A > t$, then there are no undelivered messages along x throughout the interval $[t, t_A]$.*

Proof (Lemma). Since Q is closed, we have $A \in Q$ and $\Phi_A \vdash \texttt{Active}(x)$. Since B appears blocked, we must have $\Phi_A \vdash \texttt{Sent}(x, n)$ and $\Phi_B \vdash \texttt{Received}(x, n)$ for some n.

Suppose $t_A > t$. Since $\Phi_A \vdash \texttt{Active}(x)$, x is not deactivated and not released at t_A or t. Hence, by Lemma 12, every message sent along x before t_A was received before t. Since message sends precede receipts, each of those messages was sent before t. Hence there are no undelivered messages along x throughout $[t, t_A]$.

Now suppose $t_A < t$. Since $\Phi_A \vdash \texttt{Active}(x)$, x is not deactivated and not released at t_A. By IH 2, A was blocked throughout the interval $[t_A, t]$, so it could not have sent a `release` message. Hence x is not released at t. By Lemma 12, all messages sent along x before t_A must have been delivered before t. Hence, there are no undelivered messages along x at time t. ◀

▶ **Lemma 15.** *Let $x_1 : A_1 \multimap B, \ldots, x_n : A_n \multimap B$ be a chain to $x : A \multimap B$ at time t. Then $Q \vdash \texttt{Unreleased}(x)$.*

Proof (Lemma). Since all refobs in a chain are unreleased, we know $\forall i \leq n,\ \Phi_B \nvdash \texttt{Released}(x_i)$ and so $Q \nvdash \texttt{Released}(x_i)$. It therefore suffices to prove, by induction on the length of the chain, that $\forall i \leq n,\ Q \vdash \texttt{Created}(x_i)$.

Base case: By the definition of a chain, $\alpha_t(B) \vdash \texttt{Created}(x_1)$, so $\texttt{Created}(x_1) \in \Phi_B$.

Induction step: Assume $Q \vdash \texttt{Unreleased}(x_i)$, which implies $A_i \in Q$. Let t_i be the time of A_i's snapshot.

By the definition of a chain, either the message $[B \triangleleft \texttt{info}(x_i, x_{i+1})]$ is in transit at time t, or $\alpha_t(A_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$. But the first case is impossible by Lemma 14, so we only need to consider the latter.

Suppose $t_i > t$. Lemma 14 implies that A_i cannot perform the $\textsc{SendInfo}(x_i, x_{i+1}, A_{i+1}, B)$ event during $[t, t_i]$. Hence $\alpha_{t_i}(A_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$, so $Q \vdash \texttt{Created}(x_{i+1})$.

Now suppose $t_i < t$. By IH 2, A_i must have been blocked throughout the interval $[t_i, t]$. Hence A_i could not have created any refobs during this interval, so x_{i+1} must have been created before t_i. This implies $\alpha_{t_i}(A_i) \vdash \texttt{CreatedUsing}(x_i, x_{i+1})$ and therefore $Q \vdash \texttt{Created}(x_{i+1})$. ◀

Lemma 15 implies that B cannot be in the root set. If it were, then by the Chain Lemma there would be a refob $y : C \multimap B$ with a chain where C is an external actor. Since $Q \vdash \texttt{Unreleased}(y)$, there would need to be a snapshot from C in Q – but external actors do not take snapshots, so this is impossible.

Since B is not in the root set, there must be a chain to every unreleased refob $x : A \multimap B$. By Lemma 15, $Q \vdash \texttt{Unreleased}(x)$. By Lemma 14, there are no undelivered messages to B along x at time t. Since B can only have undelivered messages along unreleased refobs (Lemma 10), the actor is indeed blocked.

Send$(x, \vec{y}, \vec{z}, A, B, \vec{C})$. In order to maintain IH 2, we must show that if $B \in Q_t$ then this event cannot occur. So suppose $B \in Q_t$. By IH 1, we must have $Q \vdash \texttt{Unreleased}(x : A \multimap B)$, so $A \in Q$. By IH 2, we moreover have $A \notin Q_t$ – otherwise A would be blocked and unable to send this message. Since B appears blocked in Q, we must have $\Phi_A \vdash \texttt{Sent}(x, n)$ and $\Phi_B \vdash \texttt{Received}(x, n)$ for some n. Since x is not deactivated at t_A and unreleased at t_B, Lemma 12 implies that every message sent before t_A is received before t_B. Hence A cannot send this message to B because $t_A > t > t_B$.

In order to maintain IH 1, suppose that one of the refobs sent to B in this step is $z : B \multimap C$, where $C \in Q_t$. Then in the next configuration, $\texttt{CreatedUsing}(y, z)$ occurs in A's knowledge set. By the same argument as above, $A \in Q \setminus Q_t$ and $\Phi_A \vdash \texttt{Sent}(y, n)$ and $\Phi_C \vdash \texttt{Received}(y, n)$ for some n. Hence A cannot perform the $\textsc{SendInfo}(y, z, A, B, C)$ event before t_A, so $\Phi_A \vdash \texttt{CreatedUsing}(y, z)$ and $Q \vdash \texttt{Created}(z)$.

SendInfo(y,z,A,B,C). By the same argument as above, $A \notin Q_t$ cannot send an `info` message to $B \in Q_t$ without violating message counts, so IH 2 is preserved.

SendRelease(x, A, B). Suppose that $A \notin Q_t$ and $B \in Q_t$. By IH 1, $x : A \multimap B$ is unreleased at time t. Since Q is finalized, $\Phi_A \vdash \texttt{Active}(x)$. Hence A cannot deactivate x and IH 2 is preserved.

In(A, R). Since every potential inverse acquaintance of an actor in Q_t is also in Q, none of the actors in Q_t is a receptionist. Hence this rule does not affect the invariants.

Out(x, B, R). Suppose $(y : B \multimap C) \in R$ where $C \in Q_t$. Then y is unreleased and $Q \vdash \texttt{Unreleased}(y)$ and $B \in Q$. But this is impossible because external actors do not take snapshots. ◀