Actor Services
Modular Verification of Message Passing Programs

Alexander J. Summers and Peter Müller

Department of Computer Science, ETH Zurich, Switzerland
alexander.summers@inf.ethz.ch, peter.mueller@inf.ethz.ch

Abstract. We present actor services: a novel program logic for defining and verifying response and functional properties of programs which communicate via asynchronous messaging. Actor services can specify how parts of a program respond to messages, both in terms of guaranteed future messages, and relations between the program states in which messages are received and responses sent. These specifications can be composed, so that end-to-end behaviours of parts of a system can be summarised and reasoned about modularly. We provide inference rules for guaranteeing these properties about future execution states without introducing explicit traces or temporal logics.
Actor services are ultimately derived from local actor services, which express behaviours of single message handlers. We provide a proof system for verifying local services against an implementation, using a novel notion of obligations to encode the appropriate liveness requirements.
Our proof technique ensures that, under weak assumptions about the underlying system (messages may be reordered, but are never lost), as well as termination of individual message handlers, actor services will guarantee suitable liveness properties about a program, which can be augmented by rich functional properties. Our approach supports reasoning about both state kept local to an actor (as in a pure actor model), and shared state passed between actors, using a flexible combination of permissions, immutability and two-state invariants.

1 Introduction

The actor model [19] is a popular programming paradigm, which structures a program execution into independent units (actors) that communicate via asynchronous messaging. This programming style was initially adopted for distributed systems [40], but has been increasingly used to develop concurrent programs, even those intended to run on a single machine. Although some actor languages support blocking (waiting) for messages, others handle message receive implicitly via built-in event loops; programming purely in this latter style eliminates the possibility of deadlocks.

Modular specification and verification of actor programs is difficult for several reasons. (1) The intended functionality is often provided by a collaboration of several communicating actors, such that the result of a computation might not be sent by the same actor to which the request was sent. This makes it difficult to relate the two messages, for instance, to express the result in terms of the request's arguments. (2) The behaviour of an actor system depends on the state of the individual actors (their call stack, e.g. [40], or the heap, e.g. [27]). However, since this state is local to an actor, it cannot be directly used to specify the behaviour for clients. (3) The local state of an actor changes dynamically in reaction to the messages it receives, for instance, to set up collaborations between actors. (4) The termination of each message handler does not ensure that senders of a message eventually receive the expected result since handlers might not send any response or send messages in circles. Therefore, actor verification requires reasoning about liveness properties.

We present (to the best of our knowledge) the first technique for actor specification and verification which solves all of these problems while supporting modular reasoning. Existing works typically either rely on a notion of whole program execution or traces, or do not handle the liveness properties needed to guarantee responsiveness; we provide specific comparisons in Sec. 5. By modularity, we mean that guaranteed behaviours of parts of a program can be proved and summarised without knowledge of the whole program, and that these summaries (specifications) can be further composed in order to derive different specifications at other levels of abstraction in the software. Support for compositional reasoning of this kind is crucial for scalability and for the reuse of verified components. We make the following main contributions. Our technique:

1. allows one to prove both response and functional properties modularly. The key idea is to introduce *actor services*, a novel state-based assertion whose validity in a state expresses that in all future states each message sent to an actor will trigger a specified response. We present a program logic that can prove these assertions modularly, without resorting to trace-based or whole-program reasoning.
2. allows one to verify actor programs at the level of the source code, rather than an abstraction to e.g. message protocols. This is enabled by a Hoare logic whose assertion language includes actor services. The logic supports a notion of obligations to express which messages must eventually be sent.
3. supports the composition of actor services to summarise behaviours of collaborating actors, without exposing these actors' existence or role in the collaboration. These summarised behaviours can be further composed.
4. allows one to specify and verify code that dynamically creates and connects actors. The behaviour of the resulting actor configuration can be specified via nested actor services.
5. supports local and shared immutable state, and permits (but does not rely on) transfer of ownership of state between actors. A permission system tracks ownership and immutability. Relational (two-state) assertions allow one to express rich functional properties on state, including both response properties and invariants on the evolution of actor-local state.

We illustrate our technique on an example from the literature, which has been the subject of previous substantial verification efforts in industry [2].

2 Programming Language and Running Example

We present our work for a simple Java-like language, in which actors are instances of special classes labelled with the **actor** keyword. These *actor classes* may declare fields, but, for simplicity, neither methods nor constructors. Instead, actor classes may declare *message handlers* prefixed with the **handler** keyword).

Actors communicate via messages. A message identifies a message handler to be invoked by its name and supplies arguments. Sending a message is an asynchronous (non-blocking) operation that enters the sent message into the recipient's message queue. After its creation, an actor enters an implicit loop. In each iteration, it *receives* a message, removing one message from its message queue and executing the corresponding message handler (or blocks if there are no queued messages). We assume that a type system ensures there is a handler for each sent message with appropriately-typed arguments. We do not assume that messages arrive in order, but require that message receive is weakly fair: if an actor continues to receive messages then each message will eventually be received. We assume that messages neither get lost nor duplicated in transit.

The local state of an actor can include heap data structures. Our technique allows multiple actors to execute in the same or in different address spaces. Our techniques are formalised such that all persistent state belongs to a (single) heap. However, we can model disjoint memories by enforcing that actors own disjoint regions of this heap, and that ownership is never transferred. Note that even in a functional language such as Erlang, the response behaviours of an actor depend on actor-local state, in terms of the actor's call stack and current stack-frame values.

2.1 The Mnesia distributed database query manager

Our running example is a protocol from a distributed database query manager called Mnesia, by Ericsson [26]. Our implementation (Fig. 1) closely follows the Erlang code [2], but actor-local data is stored in the fields of an actor.

The query protocol works as follows. When a user sends a query to the database manager (via query_setup), the query is broken down into several subqueries to be processed on different physical machines. The manager creates a *worker actor* for each subquery. The manager and worker actors are set up in a ring structure: each actor points to its successor via its next field. Once the ring of workers is set up, the manager sends a ready message to the user.

When the user receives the ready message, it (or another actor) can send a req message to the manager, which specifies the number of solutions that the user requires. This message triggers query processing by sending a sols message to the first worker in the ring. Each worker performs some local computation and then sends partial results on to the next actor, which combines them with their local computation and continues. To limit the volume of data being sent over the network, the number of results in a message is bounded by a given packet size. When a worker actor computes more than this number of solutions, it caches the remainder locally. The query manager at the head of the ring also

```
actor trait User {
  handler ready(QueryManager m);
  handler response(seq<Solution> solutions);
}

actor trait class RingParticipant {
  seq<Solution> store;
  RingParticipant next;

  handler sols(seq<Solution> solutions, int packetSize);
}

actor class QueryManager extends RingParticipant {
  User user;
  int nrSolutions;

  handler query_setup(Query query, User u) {
    RingParticipant nextPid := this;
    seq<Query> subqueries := // break down query - subqueries is non-empty
    for(int i := 0; i < |subqueries|; i := i + 1) {
      nextPid := spawn QueryWorker(next := nextPid,
                           localQuery := subqueries[i], store := []);
    }
    this.next := nextPid;
    u.ready(this);
  }

  handler req(User userPid, int needed, int packetSize) {
    this.user := userPid;
    this.nrSolutions := needed;
    this.next.sols([], packetSize);
  }

  handler sols(seq<Solution> solutions, int packetSize) {
    if(this.next != null && this.user != null) { // already initialised
      seq<Solution> newStore := solutions ++ this.store;
      if(|solutions| = 0 || |newStore| >= this.nrSolutions) {
        this.user.response(newStore);
      } else {
        this.next.sols([], packetSize);
        this.store := newStore;
      }
    }
  }
}

actor class QueryWorker extends RingParticipant {
  Query localQuery;

  handler sols(seq<Solution> solutions, int packetSize) {
    if(|this.store| >= packetSize) {
      this.next.sols(take(packetSize,this.store), packetSize);
      this.store := drop(packetSize, this.store) ++
                    filter(solutions, this.localQuery);
    } else {
      seq<Solution> newStore := this.store ++
                                filter(solutions, this.localQuery);
      this.next.sols(take(packetSize, newStore), packetSize);
      this.store := drop(packetSize, newStore);
    }
  }
}
```

Fig. 1. The running example. We assume a built-in value type **seq** for sequences with the usual operations. The **spawn** statement creates a new actor and initialises its fields. The filter operation applies the worker's local subquery to the previous worker's results; we elide the details of this database computation.

maintains a store of solutions. When it receives a sols message, it adds the received solutions to the stored solutions. If it has enough solutions to satisfy the user's request or if the sols message does not contain any solutions, the manager returns its solutions to the user (via a response message). Otherwise, it requests more solutions by sending a further sols message around the ring.

Arts and Dam [2] applied a combination of custom automated techniques and substantial manual proof effort to verify the property that: when a query is made, the user will eventually receive some response. In the remainder of this paper, we will introduce our reasoning techniques for verifying such properties.

3 Reasoning with Actor Services

An *actor service* is a novel kind of assertion that describes the consequences of a message sent to a given actor, both in terms of consequent messages that will be sent, and functional properties that will be guaranteed. An actor service consists of a left-hand side message, called the *trigger message*, and a right-hand side *response pattern*, describing possible *response messages* and additional guarantees. For example, $x.m() \rightsquigarrow y.n()$ is an actor service (and therefore an assertion in our program logic), in which $x.m()$ is the trigger message, and $y.n()$ makes up the response pattern. The meaning of this actor service is that, from the current program state, all future m messages received by the actor x are guaranteed to result in an n message being sent to the actor y. An actor service expresses a stylised form of temporal property (a *response property*), without explicitly requiring temporal connectives in the assertion logic: this liveness property is formally guaranteed provided that all message handlers terminate (see Sec. 4.5). Proving termination of such code is orthogonal; actor services guarantee that a message handler cannot terminate without sending a message leading (directly or via a sequence of further messages) to a prescribed response message, and that such a sequence of messages is guaranteed to be finite.

3.1 Actor service instantiation and composition

The essential building blocks for our actor service reasoning, are *local services.* These are actor services which can be proved against the implementation of a single message handler; in particular, the response message of such an actor service must be guaranteed to be sent during execution of the message handler for the trigger message. In Sec. 4 we will describe the details of our proof technique for verifying local services against the implementation of a message handler; for the moment, we will describe the justification of local services informally. Consider the query_setup message from the actor class QueryManager in Fig. 1. A simple example of a local service which can be proved against this implementation, is:

$$\underline{M}.\mathsf{query\_setup}(\underline{Q}, \underline{U}) \rightsquigarrow U.\mathsf{ready}(M) \qquad (QM1)$$

As a notational shorthand, we use underlined, capitalised variable names to implicitly indicate universal quantification across the actor service. This local service therefore represents that whenever any query_setup message is received by any QueryManager instance M (with any parameters Q and U), the code of the corresponding message handler will ensure that a ready message is sent to the User actor U. This property can be readily checked against the implementation of query_setup in Fig. 1.

All local services are quantified over the receiving actor and parameters; their meaning concerns all possible invocations of the message handler, and is independent of the program state. On the other hand, we can *instantiate* the local service (*QM1*) with respect to specific actors. Suppose, for example, that at a particular program point, program variable m is known to refer to a QueryManager actor, while u is known to refer to a User actor. We can instantiate (*QM1*) to derive the actor service $\mathsf{m.query\_setup}(\underline{Q}, \mathsf{u}) \rightsquigarrow \mathsf{u.ready}(\mathsf{m})$. Note that this actor service describes a property specific to these two actors, and its truth depends on the program state.

A crucial aspect of our actor service reasoning is that we can *compose* actor services to derive new ones. Suppose that the User instance u is programmed to respond to ready messages by sending *some* req message to the corresponding QueryManager. In the case of m, we can express this fact via the following actor service (we use _ to denote arguments whose values are not relevant): $\mathsf{u.ready}(\mathsf{m}) \rightsquigarrow \mathsf{m.req}(\_,\_,\_)$ This actor service can now be combined with that above, to derive: $\mathsf{m.query\_setup}(\underline{Q}, \mathsf{u}) \rightsquigarrow \mathsf{m.req}(\_,\_,\_)$. Intuitively, this derivation "chains together" the two response properties, summarising their overall guaranteed behaviour. The derived actor service still describes a response property specific to these two actors; it might not be true when User actors other than u are passed in a query_setup message.

3.2 Heap dependent expressions

Actor service composition is simple in situations such as that described above, in which all relevant expressions are (program or quantified) variables. However, to allow actor services to describe properties dependent on an actor's state (and the program heap in general), we also allow actor services to include heap dependent expressions, such as field dereferences. Consider the message handler for sols in the QueryWorker actor class. In terms of guaranteed messaging behaviour, it is clear that every sols message received will result in a further sols message being passed to the **this**.next actor. Indeed, we can derive the following local service because both branches of QueryWorker's sols handler send the required response:

$$\underline{W}.\,\mathsf{sols}\,(\underline{S}, \underline{P}) \rightsquigarrow W.\mathsf{next}.\,\mathsf{sols}\,(\_, P) \qquad (QW)$$

Heap dependent expressions in response patterns, such as $W.$next in this example, refer to the program heap when the response message is sent. This allows actor services to describe behaviours in terms of fields whose values might be appropriately set in response to the trigger message of the actor service (for example, when actors are initialised via messages). However, this interpretation means that actor service composition becomes more subtle to handle soundly.

For example, suppose that at some program point we have two $\mathsf{QueryWorker}$ instances x and y in scope, and we know that $\mathsf{x.next} = \mathsf{y}$. Instantiating the local service (QW) above for x and y, we obtain the actor services:

$$\mathsf{x.\,sols}\,(\underline{S}, \underline{P}) \rightsquigarrow \mathsf{x.next.\,sols}\,(\_, P) \quad \text{and} \quad \mathsf{y.\,sols}\,(\underline{S'}, \underline{P'}) \rightsquigarrow \mathsf{y.next.\,sols}\,(\_, P')$$

It seems that we should be able to compose these actor services, in a similar way to in the previous subsection. But we must be careful: the first actor service guarantees that, as a consequence of receiving a sols message, the actor x will send a further sols message to the actor referred to by $\mathsf{x.next}$ *at that time*. Based on these actor services alone, we do not have enough information to deduce whether composing the two would be sound; the equality $\mathsf{x.next} = \mathsf{y}$ might not hold at the relevant future points in the program execution.

Examining the code more carefully, it becomes apparent that the fields of $\mathsf{QueryWorker}$ actors are not, in fact, mutable state. They are never modified by the code, and are only set when the actors are first spawned. Immutability is a commonly-used feature in such concurrent settings, and we build in native support for immutability and other invariants of actors, as described in the following subsections. Immutability ensures soundness of the above composition.

3.3 Permissions, immutability and future states

We organise reasoning about the program heap around the notions of *ownership* and *immutability*. We model these notions formally using a permission-based logic, in the style of *implicit dynamic frames* [35]. The resulting reasoning about ownership of heap locations is closely related to verification in separation logics [28, 31], and has been used in other concurrency paradigms [24]. We employ two types of permission in our work. The standard *exclusive permission*, denoted by an assertion $\textbf{acc}(e.f)$, represents exclusive ownership of the heap location $e.f$ and permits read and write access. We also employ a notion of *immutable permissions* $\textbf{immut}(e.f)$, which permit read access only and guarantee that $e.f$ will *never* be modified. Immutable permissions are different from fractional permissions [5] because they guarantee that a location will remain immutable for the rest of the program execution. Since concurrent accesses to immutable state need not be restricted, immutable permissions may be freely duplicated, in contrast to exclusive permissions. Neither kind of permission subsumes the other. However, our logic permits exchanging an exclusive permission for a corresponding immutable permission, effectively *freezing* that location's value and making it safe for actors to concurrently access it in future. Note that permissions are a verification-only concept; they do not need to be represented at runtime.

Incorporating immutability into our reasoning about actor services is extremely powerful. In particular, any properties known to hold in the current program state which depend *only* on heap locations which are known to be immutable, may be automatically assumed to hold in all future states. We reflect this formally in our approach via a *future states relation*, written $\prec$, which reflects the semantics of immutable permissions: for any locations to which immutable permission is held *now*, immutable permission may be assumed to be held in the future, and the corresponding heap value may not have changed. Additionally, our semantics for actor services guarantees that once an actor service is true, it is also true in all future states. This design decision comes with restrictions (we do not handle explicit deallocation of actors in this paper), but allows for actor service composition without precise knowledge about program traces.

3.4 Actor invariants and message preconditions

Exclusive permissions can be used to define the parts of the heap are owned by an actor. We represent this formally using an *actor invariant.* Similar to the classical notion of object invariant [25, 14], an actor invariant is a property which must hold in between the actor's execution of each message handler. In particular, the actor's invariant may be assumed to hold at the beginning of executing the message handler, and must be shown to be re-established by the end of this execution. Both exclusive and immutable permissions may be included in an actor invariant; in the former case, this prescribes that the actor currently owns this data; in the latter, the data is immutable and may be safely shared among actors. For example, the store field of a QueryManager actor is mutated on receipt of sols messages: this can be permitted by including the exclusive permission to this field location in the actor's invariant. On the other hand, based on the observation that the next fields of QueryWorker instances are never modified, we can include **immut**(**this**.next) in the actor invariant.

Actor invariants may also include *two-state* assertions, describing constraints on the data to which permission is held. These two-state assertions can express constraints over the pairs of heaps when a message handler begins executing and when it terminates. For example, we can express in our running example that the store field of the QueryManager actor never decreases in size, using a two-state assertion $\textbf{old}(|\textbf{this}.\mathsf{store}|) \leq |\textbf{this}.\mathsf{store}|$. We use the "**old**" keyword to wrap expressions which are to be evaluated in the earlier heap.

Actor invariants must be *self-framing* (written $\models_{\mathit{frm}} A$), meaning that they depend only on heap locations to which they also require permission. For an actor invariant to include the two-state assertion above we must also include $\textbf{acc}(\textbf{this}.\mathsf{store}) * \textbf{old}(\textbf{acc}(\textbf{this}.\mathsf{store}))$. Here, the conjunction $*$ requires the permissions in both conjuncts [28]. Self-framedness guarantees that the invariants cannot be violated by other actors; all relevant heap locations are either immutable or currently owned by the actor. Actor invariants must also be *transitive* as two-state predicates: the combination of these two restrictions means that a correct actor invariant can be soundly assumed to hold across execution points spanning any number of complete message handler executions by the actor.

Our technique allows ownership of heap data to be *transferred* between actors. We prescribe that ownership of a heap location is transferred with a message, by including exclusive permission to the heap location in the precondition of the message handler. The sender may not access such a location after sending the message. As is standard, *message preconditions* (which must be self-framing) are assumed when proving properties of the message handler implementations.

3.5 Unbounded composition: summarising the ring

We can now turn to the first serious step in the proof of our running example. Let us consider the code which sets up the ring of workers, in lines 20–24 of Fig. 1. In particular, we aim to prove an actor service describing the behaviour of the actor ring, using the local service (QW) as an *actor service assumption* in our proof. In the semantics of our logic, we parameterise judgements by an *actor service environment* Λ: the set of assumed actor services. For one-state assertions in our logic, our semantic judgement has the form $\Lambda, \Sigma, \sigma \models a$, in which a is a one-state assertion, Λ is an actor service environment, Σ is a *heap-state*, consisting of a heap plus sets of exclusive and immutable permissions, and σ is a mapping from variables to values. Despite representing properties about future executions, actor services are one-state assertions in our logic: whether an actor service is true or not *now* is a well-defined property regarding the behaviour of the system from now onwards[1]. The future states relation $\Sigma_1 \prec \Sigma_2$ (Sec. 3.3) holds iff for all heap locations to which Σ_1 has immutable permission, Σ_2 also has immutable permission, and the values of the heap location are the same in the two heap-states; the values of other heap locations are unconstrained.

To prove the behaviour of a loop, we require an invariant; in our technique this can include actor services. The following assertion suffices for our example:

$$\begin{gathered}\mathsf{nextPid} \neq \mathbf{null} \ \wedge \ 0 \leq \mathsf{i} \leq |\mathsf{subqueries}| \ \wedge \ (\mathsf{i} = 0 \ \Rightarrow \ \mathsf{nextPid} = \mathbf{this}) \wedge \\ (\mathsf{i} > 0 \ \Rightarrow \ (\mathsf{nextPid}.\mathsf{sols}(\underline{S}, \underline{P}) \rightsquigarrow (\mathbf{this}.\mathsf{sols}(\_, P)))\end{gathered}$$

This actor service expresses that the part of the ring built so far guarantees that sending a sols message to the last actor created will cause a corresponding sols message to be eventually sent to the **this** actor. Intuitively, this is because each actor in the ring promises—via the local service (QW)—to send such a message to the next actor in the ring, and these next fields reach **this**.

Establishing this loop invariant before the loop is uninteresting, as no actor services are required. We focus on how to justify that the loop invariant is preserved, in particular, how we derive the required actor service at the end of the loop body. Let us begin with the (simpler) case in which i is initially 0 before executing the loop body. We can instantiate the assumed actor service (QW) with nextPid as the receiver of the trigger message, to obtain the actor service $\mathsf{nextPid}.\mathsf{sols}(\underline{S}, \underline{P}) \rightsquigarrow \mathsf{nextPid}.\mathsf{next}.\mathsf{sols}(\_, \underline{P})$. We now consider the rule for rewriting the response patterns of actor services; the following is a simplified version of the full rule:

$$\frac{\begin{gathered}\Lambda, \Sigma, \sigma \models T \rightsquigarrow e.m(\overrightarrow{e_i}) \\ \forall \Sigma_1.\ \Sigma \prec \Sigma_1 \ \Rightarrow \ \Lambda, \Sigma_1, \sigma \models e = e' * (\ast\, \overrightarrow{e_i = e'_i})\end{gathered}}{\Lambda, \Sigma, \sigma \models T \rightsquigarrow e'.m(\overrightarrow{e'_i})} \ (\textit{rewrite-simple})$$

[1] Our analogous judgement for two-state assertions takes a second heap-state, and is written $\Lambda, \Sigma_1, \Sigma_2, \sigma \models A$. We also employ a judgement $\Sigma, \sigma \models_{\mathit{frm}} e$ (e is *framed* in the state), meaning that the expression e depends only on heap locations to which permission is held, and $\Sigma, \sigma \models_{\mathit{immut}} e$ to express the same restricted to immutable locations (e is *immutable* in this state).

We use the metavariable T to range over trigger messages (i.e., the left-hand-sides of actor services). The conjunction $*$ used in our formalisation is equivalent to standard logical conjunction ($\wedge$) when applied to assertions without permissions, such as these. We use the notation ($\overrightarrow{* \, e_i = e_i'}$) to represent iterated conjunction over each $e_i = e_i'$ assertion. This rule expresses that we can rewrite the expressions used in the response pattern of an actor service via equalities which can be shown to hold in *all* future states (according to the $\prec$ relation introduced in Sec. 3.3). In practice, this premise can be satisfied only if the equalities are either trivial (on identical expressions), or are known to hold in the current state, and which depend only on immutable heap locations[2].

When we create a new QueryWorker instance (on line 20), we obtain exclusive permission to the fields of the new actor. By choosing to logically *freeze* (i.e., exchange the exclusive permission for immutable permission) the location nextPid.next at the point of spawning the actor, not only are we able to establish immutable permission for the actor's invariant, but we can deduce that the equality nextPid.next = **this** will indeed hold in all future states. Using the (*rewrite-simple*) rule above, we can therefore obtain the actor service required in the loop invariant.

Now let us turn to the case in which i is greater than 0 before executing the loop body. In this case, we can apply similar reasoning to obtain an actor service describing the behaviour of the newly-spawned actor; in order to establish the loop invariant we need to compose this actor service with the one from the loop invariant assumed initially. We can now present (again, a simplified form of) the rule for composing two actor services:

$$\frac{\begin{array}{c}\Lambda, \Sigma, \sigma \models T \rightsquigarrow e.m(\overrightarrow{e_i}) \\ \forall \Sigma_1.\ \Sigma \prec \Sigma_1 \ \Rightarrow\ \Lambda, \Sigma_1, \sigma \models e.m(\overrightarrow{e_i}) \rightsquigarrow R\end{array}}{\Lambda, \Sigma, \sigma \models T \rightsquigarrow R} \ (\textit{compose-simple})$$

We use the metavariable R to range over response patterns (i.e., the right-hand-sides of actor services). The second premise requires that the specified actor service will hold in any future state. Based on the technique introduced so far, there are two ways to establish this. Firstly, some actor services can be derived from the assumed actor services in Λ. Secondly, actor services known to hold in the current state can also be assumed to hold in all future states (as described by $\prec$), provided the expressions used in their trigger messages are immutable.

Returning to our example, the loop invariant provides us with the actor service $\mathsf{nextPid}_0.\mathsf{sols}\,(\underline{S}, \underline{P}) \rightsquigarrow$ **this**$.\mathsf{sols}\,(\_, P)$ at the start of the loop iteration (we write $\mathsf{nextPid}_0$ for the value of nextPid at this point). By similar reasoning to in the i$\,=0$ case, we are able to obtain the actor service $\mathsf{nextPid}.\mathsf{sols}\,(\underline{S}, \underline{P}) \rightsquigarrow \mathsf{nextPid}_0.\mathsf{sols}\,(\_, P)$. We can then *compose* these two actor services, to obtain[3] the desired actor service $\mathsf{nextPid}.\mathsf{sols}\,(\underline{S}, \underline{P}) \rightsquigarrow$ **this**$.\mathsf{sols}\,(\_, P)$.

[2] The general form of this rule, however, allows for more involved cases; see Sec. 3.9.

[3] We elide the handling of the quantified variables here, but they are instantiated and generalised in a standard way, as formalised later in Fig. 2.

This simple case of composition essentially matches one response message in one actor service with the trigger message with another, and allows us to deduce a service in which this intermediate message is hidden, summarising the end-to-end behaviour of the actors. After the loop (at line 24), the loop invariant, the negation of the loop guard, and the fact that $\mathsf{subqueries}$ is non-empty imply $\mathsf{nextPid.sols}\,(\underline{S}, \underline{P}) \leadsto \textbf{this}.\mathsf{sols}\,(\_, P)$. After the assignment $\textbf{this}.\mathsf{next} := \mathsf{nextPid}$ on line 24, which completes the ring of actors, we can deduce this actor service with $\textbf{this}.\mathsf{next}$ as the receiver of the trigger message instead of $\mathsf{nextPid}$:

$$\textbf{this}.\mathsf{next.sols}\,(\underline{S}, \underline{P}) \leadsto \textbf{this}.\mathsf{sols}\,(\_, P) \tag{1}$$

This actor service represents a responsiveness property that is justified by the whole ring of actors created, but without revealing their number or underlying structure. In the rest of this section, we will show how our general reasoning technique allows us to further combine actor services with this one, to obtain actor services to describe the example as a whole.

3.6 General actor services

As well as specifying response properties in terms of guaranteed messages, it is important to specify and verify functional properties associated with these responses. For example, in the case of the req message in the $\mathsf{QueryManager}$ class, the $\textbf{this}.\mathsf{user}$ field will be set by the message handler, and never modified (we can consider the location immutable, from this point onwards). This fact is relevant for later reasoning about the $\mathsf{response}$ message eventually sent to this User actor.

It is also important for our response properties to be able to describe multiple *alternative* responses, as well as conditions under which they may be known to be individually guaranteed. For example, the behaviour of the sols message handler in $\mathsf{QueryManager}$ cannot be simply summarised by a single response message.

We achieve these goals with two complementary features: response patterns with multiple alternatives, and *where-clauses*, which describe additional properties guaranteed when response messages are sent.

Definition 1 (Actor Services). *An actor service is an assertion of the form $\forall \overrightarrow{X_j}.(T \leadsto R)$, where T is a* trigger message*, and R a* response pattern[4].
*A trigger message T, is a term $e.m(\overrightarrow{e_i})$, where m is a message name, and $e, \overrightarrow{e_i}$ are one-state expressions (i.e. do not mention **old**).*
Response patterns (ranged over by R), are finite sets of responses*; we notate response patterns in examples as $(r_1 \mid r_2 \mid \ldots \mid r_n)$.*
A response *r is a response message or empty response.*
A response message *has the form $(e.m(\overrightarrow{e_i})$ where $A)$, in which A is a two-state assertion (i.e., may include **old**), called the* where-clause*.*
An empty response *has the form $(\epsilon$ where $\textbf{old}(a))$, for a one-state assertion a.*
*In both cases, where-clauses may not mention exclusive (**acc**) permissions, and an omitted where-clause is the same as writing where true.*

[4] In examples, we omit the explicit quantifiers, and use (as previously) the $\underline{X}$ notation.

The meaning of an actor service with response pattern R, is that, for all trigger messages in the future, *at least one* of the cases described by the response pattern is guaranteed to eventually happen, and its where-clause will be guaranteed to hold at that point. An empty response permits that no message will be sent; this can be used to handle special cases in the actor's behaviour, or simply to weaken the meaning of an actor service. In all cases, the "**old**" heap in a where-clause refers to the heap when the trigger message was received; the two-state where-clauses can thus relate this state with the state in which response messages are sent (in the case of an empty response, there is no such state, hence the restriction to the "**old**" state). We show examples in the following subsections.

3.7 Where-clauses and composition

Where-clauses allow actor services to express functional properties, beyond those guaranteed by message preconditions. For example, the following local service[5] expresses the relevant behaviour of the req message handler:

$$\underline{M}.\mathsf{req}(\underline{U},\underline{N},\underline{P}) \rightsquigarrow M.\mathsf{next}.\,\mathsf{sols}\,(\_,P)\ \textit{where}\ \mathbf{immut}(M.\mathsf{user}) * M.\mathsf{user} = U \qquad (\textit{QM2})$$

As with all heap-dependent expressions in our logic, where-clauses may describe properties of heap locations only when appropriate permissions are held. In the case of a where-clause, these can either be immutable permissions included in the where-clause itself, or permissions guaranteed by the preconditions of the corresponding trigger message (in the "**old**" state of the where-clause) and response message. Such where-clauses can include additional information about the state passed around with messages, which may be true for the guaranteed response messages but not *all* messages of this kind.

Since where-clauses can only constrain state which is framed by permissions associated with the corresponding messages, their meaning is stable whether considered with respect to when those messages were sent or when they begin being handled. This allows us to extend actor service composition to "chain together" where-clauses into two-state assertions summarising their transitive guarantees:

Definition 2 (Three state composition). *We define that* A_3 *is a* three-state combination *of assertions* A_1 *and* A_2 *with respect to a current state, via the predicate* $(\Lambda,\Sigma,\sigma).\mathit{futureCombines}(A_1,A_2,A_3)$*, which holds iff:*
$\forall \Sigma_1,\Sigma_2,\Sigma_3.\ \Sigma{\prec}\Sigma_1{\prec}\Sigma_2{\prec}\Sigma_3 \ \Rightarrow$
$(\Lambda,\Sigma_1,\Sigma_2,\sigma \models A_1 \text{ and } \Lambda,\Sigma_2,\Sigma_3,\sigma \models A_2 \ \Rightarrow\ \Lambda,\Sigma_1,\Sigma_3,\sigma \models A_3)$

We generalise this notion to a predicate on a single two-state assertion and two response patterns, written $(\Lambda,\Sigma,\sigma).\mathit{futureCombines}(A,R,R')$*, which holds if* R' *is the same response pattern as* R *except that each where-clause in* R' *is a three-state combination of* A *and the corresponding where-clause in* R*.*

[5] Recall that a local actor service is one provable with respect to the code of the corresponding message handler, and which is true in all states.

$$\frac{(T \rightsquigarrow R) \in \Lambda}{\Lambda, \Sigma, \sigma \models T \rightsquigarrow R}\;(\mathit{axiom})$$

$$\frac{\begin{array}{c}\Lambda, \Sigma, \sigma \models T \rightsquigarrow ((e.m(\overrightarrow{e_i})\ \mathit{where}\ A) \cup R)\\ (\Lambda, \Sigma, \sigma).\mathit{futureEntails}(A, e.m(\overrightarrow{e_i}) \rightsquigarrow R')\\ (\Lambda, \Sigma, \sigma).\mathit{futureCombines}(A, R', R'')\end{array}}{\Lambda, \Sigma, \sigma \models T \rightsquigarrow (R'' \cup R)}\;(\mathit{compose})$$

$$\frac{\begin{array}{c}\Lambda, \Sigma, \sigma \models e.m(\overrightarrow{e_i}) \rightsquigarrow ((e'.m'(\overrightarrow{e'_j})\ \mathit{where}\ A') \cup R)\\ X, \overrightarrow{X_i} \notin \mathit{dom}(\sigma) \quad \sigma' = \sigma[X \mapsto \llbracket e \rrbracket_{\Sigma,\sigma}][X_i \mapsto \llbracket e_i \rrbracket_{\Sigma,\sigma}]\\ A_m \;=\; \textbf{old}(\mathit{pre}(m, X, \overrightarrow{X_i})) \quad A \;=\; A_m * \mathit{pre}(m', e', e'_j)\\ (\Lambda, \Sigma, \sigma').\mathit{futureEntails}(A * A', A * A'' * e'{=}e'' * (\circledast\, e'_j{=}e''_j))\end{array}}{\Lambda, \Sigma, \sigma \models e.m(\overrightarrow{e_i}) \rightsquigarrow ((e''.m'(\overrightarrow{e''_j})\ \mathit{where}\ A'') \cup R)}\;(\mathit{rewrite})$$

$$\frac{\Lambda, \Sigma, \sigma \models T \rightsquigarrow ((e_1.m_1(e'_1)\ \mathit{where}\ \mathit{false}) \cup R)}{\Lambda, \Sigma, \sigma \models T \rightsquigarrow R}\;(\mathit{elimFalse})$$

$$\frac{\begin{array}{c}\Lambda, \Sigma, \sigma \models e.m(\overrightarrow{e_i}) \rightsquigarrow ((e.m(\overrightarrow{e'_i})\ \mathit{where}\ A_1) \cup R) \quad \Sigma, \sigma \models_{\mathit{immut}} e\\ (\Lambda, \Sigma, \sigma).\mathit{futureEntails}(A_1, \mathit{localVariant}(e))\end{array}}{\Lambda, \Sigma, \sigma \models e.m(\overrightarrow{e_i}) \rightsquigarrow R}\;(\mathit{localVariant})$$

$$\frac{\Lambda, \Sigma, \sigma \models \forall x.(T \rightsquigarrow R) \quad \Sigma, \sigma \models_{\mathit{frm}} e \quad x \in FV(R) \;\Rightarrow\; \Sigma, \sigma \models_{\mathit{immut}} e}{\Lambda, \Sigma, \sigma \models (T \rightsquigarrow R)[e/x]}\;(\forall E)$$

$$\frac{X \notin \mathit{dom}(\sigma) \quad \forall v.\ (\Lambda, \Sigma, \sigma[X \mapsto v] \models (T \rightsquigarrow R)[X/x])}{\Lambda, \Sigma, \sigma \models \forall x.(T \rightsquigarrow R)}\;(\forall I)$$

Fig. 2. Semantics for actor services. *pre* denotes a message precondition instantiated with receiver and parameters. $\llbracket e \rrbracket_{\Sigma,\sigma}$ denotes evaluation of the expression e in Σ, σ.

Equipped with this definition, we can now explain the general rule for composing actor services, a simplified version of which was shown in Sec. 3.5. Fig. 2 shows the full rules for deriving actor services; we consider here the rule (*compose*) (the others will be explained in the remainder of this section). Compared with the simplified version (*compose-simple*) a number of generalisations have been made. The first premise now handles the possibility of alternative response patterns R in the original actor service. The third premise prescribes that the new where-clauses in the resulting composed actor service are defined in terms of three-state combinations of A and the where-clauses in R' (the response pattern of the second actor service composed). The second premise has also been changed; the predicate $(\Lambda, \Sigma, \sigma).\mathit{futureEntails}(A_1, A_2)$ checks entailment between A_1 and A_2 in all pairs Σ_1, Σ_2 of states such that $\Sigma \prec \Sigma_1$ and $\Sigma_1 \prec \Sigma_2$. Thus, rather than requiring that the actor service in the second premise holds in *all* future states, we can use information from the where-clause A to help justify this premise.

Returning to the actor service (1), derived at line 24 in our example, if we now consider (*QM2*) an assumed actor service, we can instantiate it[6] and use the rule (*combine*) to derive the following actor service at this program point:

$$\textbf{this}.\mathsf{req}(\underline{U}, \underline{N}, \underline{P}) \leadsto \textbf{this}.\mathsf{sols}\,(\_, P) \ \textit{where}\ \textbf{immut}(\textbf{this}.\mathsf{user}) * \textbf{this}.\mathsf{user} = U \tag{2}$$

This assertion expresses the response property that a subsequent call to req on this actor will eventually cause a sols message to be received by the same actor (the first response from the ring of QueryWorker actors). We explain next how to reason about the subsequent behaviour of the ring, on receiving this message.

3.8 Local variants: reasoning about callback loops

Just as for reasoning about recursion in a sequential setting, we need extra machinery to reason about situations in which a message might result in the same message type being sent to the same actor. Assuming this behaviour is not intended to go on forever, we need a means of justifying its eventual termination. This is, however, challenging to achieve modularly, since the *reason* for termination may depend on state which is local to the actors involved.

We illustrate our solution with respect to the sols implementation in the QueryManager class, for which we require alternative responses. The most-general actor service which we prove against the implementation (i.e., it is a local service), is the following:

$$\begin{aligned}\underline{M}.\mathsf{sols}\,(\underline{S}, \underline{P}) \leadsto\ & (M.\mathsf{user}.\mathsf{response}(\_)) \\ \mid\ & (M.\mathsf{next}.\mathsf{sols}\,(\_, P)\ \textit{where}\ \textit{localVariant}(M)) \\ \mid\ & (\epsilon\ \textit{where}\ \textbf{old}(M.\mathsf{next} = \textbf{null}\ \vee\ M.\mathsf{user} = \textbf{null}))\end{aligned} \tag{\textit{QM3}}$$

This actor service specifies that there are three possibilities: the user (as read from the actor's user field) will receive a response message, the next actor (i.e., the head of the ring) will receive a sols message, asking for more solutions, or, in the case where the actor was not yet properly initialised, no response message is guaranteed.

The assertion *localVariant*(M) has not yet been introduced; this is an assertion which can *only* occur in where-clauses, and we will explain its meaning and usage in this subsection. Let us consider first composing (an instantiation of) the local service (*QM3*) above (in particular, its second response message) with the actor service (1) derived at line 24. This allows us to derive the actor service:

$$\begin{aligned}\textbf{this}.\mathsf{sols}\,(\underline{S}, \underline{P}) \leadsto\ & (\textbf{this}.\mathsf{user}.\mathsf{response}(\_)) \\ \mid\ & (\textbf{this}.\mathsf{sols}\,(\_, P)\ \textit{where}\ \textit{localVariant}(\textbf{this})) \\ \mid\ & (\epsilon\ \textit{where}\ \textbf{old}(\textbf{this}.\mathsf{next} = \textbf{null}\ \vee\ \textbf{this}.\mathsf{user} = \textbf{null}))\end{aligned} \tag{3}$$

[6] The last premise of the ($\forall E$) rule guards against the possibility of instantiating heap-dependent expressions into contexts in which they would be interpreted with respect to other (future) heaps, *unless* they are known to be immutable (trivially true for variables such as **this** which do not depend on the heap).

Considering the second case of this response pattern, the actor service describes a looping behaviour; one possibility for the response to receiving a sols message is that the actor will eventually be sent a further sols message. This correctly describes a behaviour of the protocol, but we wish to show that this second alternative will not be taken indefinitely. By manual inspection of the code (lines 37–42), we can see that this will indeed not be the case: every time the sols message handler chooses to send a further sols message to the next actor, the number of remaining required solutions will have decreased. Note that this amount can be precisely expressed only in terms of state local to the actor: by the expression **this**. nrSolutions $-$ |**this**. store|.

Our *localVariant* assertions solve this problem; notionally, they prescribe the *existence* of a variant expression (in the standard sense for termination checking) in terms of the actor-local state, which is guaranteed to satisfy the following property with respect to the **old** state (i.e., the state in which the handler for trigger message of the actor service began executing): for *all* subsequent message handlers executed by the actor, this expression will have a smaller value than it did in the **old** state. In the semantics of our logic, we do not define this intended meaning for *localVariant* assertions, which would depend on knowing future traces of the program. Instead, we treat *localVariant* assertions formally as *uninterpreted predicates* over the current states. The only property assumed for these assertions, is that once true they will be true for all future states, i.e.: $\Lambda, \Sigma_0, \Sigma_1, \sigma \models \mathit{localVariant}(x) \ \mathit{and}\ \Sigma_1 \prec \Sigma_2 \ \Rightarrow\ \Lambda, \Sigma_0, \Sigma_2, \sigma \models \mathit{localVariant}(x)$ Note that this property was necessary for deducing actor service (3), above.

Discharging the correct proof obligations to show that a local variant indeed exists for a particular implementation is non-trivial, but we handle this problem in Sec. 4. From the perspective of our actor service semantics here, the rule (*localVariant*) of Fig. 2 allows us to make use of the *existence* of a local variant, as a justification for *removing* the corresponding response message from the alternatives. This allows us to derive a stronger actor service, reflecting that at least one of the other alternatives will happen eventually:

$$\begin{array}{rl} \textbf{this}.\mathsf{sols}\,(\underline{S}, \underline{P}) \rightsquigarrow & (\textbf{this}.\mathsf{user}.\mathsf{response}(\_)) \\ | & (\epsilon\ \mathit{where}\ \textbf{old}(\textbf{this}.\mathsf{next} = \textbf{null}\ \vee\ \textbf{this}.\mathsf{user} = \textbf{null})) \end{array} \qquad (4)$$

3.9 Rewriting and eliminating alternatives

Equipped with the full rules for deriving actor services, we show now how, at the same program point in our example (line 24, after the ring has been initialised), we can derive an actor service describing the overall function of the ring. We consider first the actor service (2), and observe that the **this**.user field is guaranteed immutable and non-null, by a combination of the where-clause and the precondition of the trigger message req. We can make this explicit, by applying the general form of the (*rewrite*) rule from Fig. 2 to (2). The extra complexity in this rule allows the where-clause A' to be rewritten into a new form A'' using facts from the corresponding message preconditions: in this case, the "**old**" precondition A_m guarantees the property that the passed user parameter will be non-null. In addition, the use of *futureEntails* in this premise allows us to make use of immutable facts known at this particular program point; in particular, we can use the fact that **this**.next is non-null and immutable in the current state (after executing line 24), to rewrite the where condition further:

$$\begin{array}{l}\textbf{this}.\mathsf{req}(\underline{U},\underline{N},\underline{P}) \rightsquigarrow \textbf{this}.\mathsf{sols}(\_,P)\ \textit{where}\ \textbf{immut}(\textbf{this}.\mathsf{user}) * \\ \textbf{this}.\mathsf{user} = U * U \neq \textbf{null} * \textbf{immut}(\textbf{this}.\mathsf{next}) * \textbf{this}.\mathsf{next} \neq \textbf{null}\end{array} \tag{5}$$

We now compose the actor services (5) and (4); the facts about immutable data in the where condition of (5) can, where desired, be preserved in the resulting where-clause. In particular, these facts contradict the where-clause of the empty response in (4); we can derive the following actor service:

$$\begin{array}{l}\textbf{this}.\mathsf{req}(\underline{U},\underline{N},\underline{P}) \rightsquigarrow \\ \quad (\textbf{this}.\mathsf{user}.\mathsf{response}(\_)\ \textit{where}\ \textbf{immut}(\textbf{this}.\mathsf{user}) * \textbf{this}.\mathsf{user} = U) \\ \mid\ (\epsilon\ \textit{where false})\end{array} \tag{6}$$

Finally, we can use the rule (*rewrite*) to replace the expression **this**.user with U in the first alternative and then drop the where-clause, and the rule (*elimFalse*) to eliminate the second alternative (its where condition shows it to be an unfeasible response in this state), to derive the desired response property expressed by an actor service: $\textbf{this}.\mathsf{req}(\underline{U},\underline{N},\underline{P}) \rightsquigarrow U.\mathsf{response}(\_)$.

3.10 Nested actor services and formal semantics

With respect to our running example, we have shown how to deduce an important response property at an intermediate program point in the code of the query_setup message handler. Several of the inference steps above depended on specific properties known to hold at this program point. However, we would like to present a specification to a client of the database protocol, which will not require knowledge of this program code. We can achieve this in a natural way: we support actor services in the where-clauses of other actor services. We can then improve upon the very first actor service mentioned in this section (*QM1*), using a where-clause to describe the guaranteed functionality:

$$\underline{M}.\mathsf{query\_setup}(\underline{Q},\underline{U'}) \rightsquigarrow U'.\mathsf{ready}(M)\ \textit{where}\ M.\mathsf{req}(\underline{U},\underline{N},\underline{P}) \rightsquigarrow U.\mathsf{response}(\_) \tag{QM}$$

This precisely summarises the specification of the database manager from the client's perspective: *after* calling the query_setup message, the passed user is guaranteed a ready message in response, *by which time* the QueryManager will promise to respond to a subsequent req message with an eventual response message. Note that this specification exposes no details about the complexities of the implementation and messaging protocol; a simpler (but less efficient) implementation could satisfy the same specification.

The rules of Fig. 2 in fact define our formal semantics for actor service assertions. That is, we interpret a judgement $\Lambda, \Sigma, \sigma \models T \rightsquigarrow R$ according to the least fixpoint interpretation of these rules; equivalently, an actor service is true under actor service assumptions Λ in a state Σ, σ if there is a finite derivation of this fact, according to these rules[7]. Nested actor services can be simply handled with the same rules; in particular, it is possible to rewrite an actor service in a where-clause via the (*rewrite*) rule, just like any other assertion.

Throughout this section, we have required actor services as assumptions, which we have claimed to be *local services*: those which can be proved against the implementation of a particular message handler. In the next section, we will present our techniques for justifying these local services formally.

4 Proving Local Services

In this section, we define a proof system in the style of Hoare Logic, for proving properties about message handler implementations. The main goal is the proof of local services against such code; recall that a local service is an actor service whose meaning doesn't depend on a particular program state, and whose response messages must be sent *directly* by the code of the handler for the trigger message. Thus, one of the requirements on our design is that we can express proof obligations that require all executions of the message handler to *eventually* reach a point in which the requirements of a response pattern are satisfied. We achieve this with the novel notion of *obligation assertions*. These assertions are not to be used in specifications, and may not occur in actor services; they are used *only* during proofs in our Hoare Logic. They can, however, be used to encode the requirements that a (local) actor service imposes on a message handler implementation, as we will show. Judgements in our Hoare Logic (hereafter, Hoare triples) are of the form $\Lambda \vdash \{A_1\}\ s\ \{A_2\}$, where Λ is an actor service environment, A_1 is a self-framing two-state assertion, called the *precondition*, s is a statement, and A_2 is a self-framing two-state assertion, called the *postcondition*. As well as describing properties of the usual states (before/after execution of the statement), our two-state assertions can include facts about and relationships with a fixed "old" state, used in practice to denote the pre-state of execution of the entire message handler. This support for two-state assertions allows us, for example, to handle proof obligations about actor invariants: we can simply require the invariant of the actor in the postcondition of a judgement, to enforce the obligation that the invariant is re-established. The restriction to self-framing assertions in Hoare triples [37] guarantees that facts concerning heap values can only be preserved while appropriate permissions are known to be held. We define our Hoare Logic with respect to the following language:

[7] Nested actor services require care to ensure that this semantics is well-defined. The definition of *futureEntails* is not a simple entailment check in the case of actor service assertions. We employ a construction to guarantee that a guaranteed actor service can be added as an additional *assumption* in deriving the "entailed" formula, avoiding negative occurrences of actor services. This makes the derivation rules monotonic with respect to actor service assertions; the fixpoint is guaranteed to exist.

$$\frac{A_1 \models^{\Lambda} A_1' \quad A_2' \models^{\Lambda} A_2 \quad \Lambda \vdash \{A_1'\}\ s\ \{A_2'\}}{\Lambda \vdash \{A_1\}\ s\ \{A_2\}}\ (\mathit{cons})$$

$$\frac{\mathit{freeze}(A_1, A_1') \quad \mathit{freeze}(A_2', A_2) \quad \Lambda \vdash \{A_1'\}\ s\ \{A_2'\}}{\Lambda \vdash \{A_1\}\ s\ \{A_2\}}\ (\mathit{freeze})$$

$$\frac{FV(A_3) \cap \mathit{mods}(s) = \emptyset \quad \models_{\mathit{frm}} A_3 \quad \Lambda \vdash \{A_1\}\ s\ \{A_2\}}{\Lambda \vdash \{A_1 * A_3\}\ s\ \{A_2 * A_3\}}\ (\mathit{frame})$$

$$\frac{}{\Lambda \vdash \{A\}\ \mathtt{skip}\ \{A\}}\ (\mathit{skip}) \qquad \frac{x \notin \mathit{otherStateFV}(A) \quad A[e/x] \models_{\mathit{frm}} e}{\Lambda \vdash \{A[e/x]\}\ x{:=}e\ \{A\}}\ (\mathit{varAss})$$

$$\frac{}{\Lambda \vdash \{\mathbf{acc}(x.f)\}\ x.f{:=}y\ \{\mathbf{acc}(x.f) * x.f{=}y\}}\ (\mathit{fldAss})$$

$$\frac{\begin{array}{c} x \notin FV(A) \quad \mathit{fields}(C) = \overrightarrow{f_i} \quad \overrightarrow{A \models_{\mathit{frm}} e_i} \quad \mathit{freeze}((\circledast\, \overrightarrow{\mathbf{acc}(x.f_i)}\,), A') \\ \models_{\mathit{frm}} a \quad A * A' * (\circledast\, \overrightarrow{x.f_i = e_i}\,) \models^{\Lambda} A'' * a \\ \forall \Sigma_1, \sigma.\ (\Sigma_1, \sigma \models a \ \Rightarrow\ \exists \Sigma_0.\ \Sigma_0, \Sigma_1, \sigma \models \mathit{Inv}(C)[x/\mathbf{this}]) \end{array}}{\Lambda \vdash \{A\}\ x{:=}\ \mathtt{spawn}\ C(\overrightarrow{f_i{:=}e_i}\,)\ \{A''\}}\ (\mathit{spawn})$$

$$\frac{a = \mathit{pre}(m, e, \overrightarrow{e_i}) \quad a * A' \models_{\mathit{frm}} e \quad \overrightarrow{a * A' \models_{\mathit{frm}} e_i}}{\Lambda \vdash \{a * A'\}\ e.m(\overrightarrow{e_i})\ \{A'\}}\ (\mathit{messageNoObl})$$

$$\frac{\begin{array}{c} a = \mathit{pre}(m, e, \overrightarrow{e_i}) \quad A \models^{\Lambda} a * A' \quad (e'.m(\overrightarrow{e_i'})\ \mathit{where}\ A_1) \in R \\ A \models_{\mathit{frm}} e = e' \wedge (\bigwedge \overrightarrow{e_i = e_i'}\,) \quad A \models^{\Lambda} e = e' \wedge (\bigwedge \overrightarrow{e_i = e_i'}\,) \quad A \models_{\mathit{frm}} A_1 \\ (A_2, B_2) = \mathit{splitLocalVariant}(A_1) \quad A \models^{\Lambda} A_2 \quad A * B_2 \models^{\Lambda} A * B_3 \end{array}}{\Lambda \vdash \{A * \mathbf{obl}(R)\}\ e.m(e')\ \{A' * (B_3 \Rightarrow \mathbf{obl}(\mathit{localVariant}(\mathbf{this})))\}}\ (\mathit{messageObl})$$

$$\frac{(\epsilon\ \mathit{where}\ \mathbf{old}(a)) \in R \quad A \models^{\Lambda} \mathbf{old}(a) \quad \Lambda \vdash \{A\}\ s\ \{A'\}}{\Lambda \vdash \{A * \mathbf{obl}(R)\}\ s\ \{A'\}}\ (\mathit{emptyObl})$$

$$\frac{\begin{array}{c} C = \mathit{cls}(\mathbf{this}) \quad A * B \models^{\Lambda} a \quad \models_{\mathit{immut}} a \quad a * \mathit{Inv}(C) \models_{\mathit{frm}} e \leq \mathbf{old}(e) \\ a * \mathit{Inv}(C) \models^{\Lambda} e \leq \mathbf{old}(e) \quad A * B \models^{\Lambda} \mathbf{old}(e) \geq 0 \wedge e < \mathbf{old}(e) \end{array}}{\Lambda \vdash \{A * (B \Rightarrow \mathbf{obl}(\mathit{localVariant}(\mathbf{this})))\}\ \mathtt{end}\ \{A\}}\ (\mathit{localVarObl})$$

Fig. 3. Hoare Logic for Proving Message Handler Properties. *Inv*(C) denotes the (two-state) actor invariant for class C.

Definition 3 (Program Syntax). Statements s *are defined by the grammar:*
$s ::= \mathtt{skip} \mid x{:=}e \mid x.f{:=}y \mid x{:=}\ \mathtt{spawn}\ C(\overrightarrow{f_i{:=}e_i}\,) \mid (s_1; s_2)$
$\quad \mid\ (\mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2) \mid (\mathtt{while}\ b\ \mathtt{do}\ s_1) \mid e.m(\overrightarrow{e_i}) \mid \mathtt{end}$
Message handler bodies have the form $(s; \mathtt{end})$*, where* s *does not contain* `end`*.*

We use `end` as a syntactic marker for the end of a message handler body; this is useful for formalising requirements which can be checked only at this point.

4.1 Hoare Logic derivations and valid programs

The rules of our Hoare Logic are given in Fig. 3. The rules for if-conditions, while loops and sequential composition are standard, and we omit them. The first premise of the (*varAss*) rule requires that x occurs neither under **old** nor in the response patterns of actor services in the assertion A; this avoids the possibility of a heap dependent e being substituted into a position in which it would be evaluated in the wrong heap.

The (*freeze*) rule is similar in nature to the rule of consequence, but allows us to rewrite assertions to replace **acc** permissions with **immut** permissions. The predicate $\mathit{freeze}(A_1, A_2)$ holds if A_2 is syntactically identical to A_1 except possibly for some $\mathbf{acc}(e.f)$ subformulas of A_1 being replaced by corresponding $\mathbf{immut}(e.f)$ formulas. The actual rule of consequence (*cons*) makes use of an entailment operator ($\models^{\Lambda}$) which is standard except that the actor service environment Λ is used when checking the entailment. This notion of entailment supports arbitrary reasoning within our overall logic: in particular, it can be used to derive new actor service assertions, according to their semantics in Fig. 2[8].

The (*spawn*) rule is relatively complex, but the various premises essentially capture the following ideas. Firstly, when an actor is $\underrightarrow{\text{spawned}}$, exclusive permission to all of its fields (denoted by the assertion $(\ast\ \overline{\mathbf{acc}(x.f_i)}\ )$) is newly made available, and we can assume that all fields have been initialised to the specified e_i expressions. The actor's invariant might require some of these exclusive permissions (in which case they must be given up in the current scope); it might also require *immutable* permission to some of these fields. To handle this possibility, the *freeze* operator can be used to obtain necessary immutable permissions in A'. The (one-state) assertion a must be strong enough to guarantee a weak form of the actor's invariant, in which all that needs to be justified about the **old** state is that its existence is not inconsistent. To satisfy this premise, the assertion a needs to include the permissions which the actor invariant requires, which forces them not to also occur in A'', unless they are immutable permissions.

Even without obligations (described in the next subsection), we can now define what it means for a message implementation to be *valid*. This judgement is independent of any actor service reasoning: it guarantees that the first four properties described in our list above are all true for a particular message implementation, and define the baseline verification condition for a given program.

Definition 4 (Valid message handler). *A handler for message m in class C is* valid*, written $C, m \vdash_{OK}$, iff (where a_{pre} is the precondition of m and s_{body} is the body of message m) there exist a one-state assertion a and expressions $\overrightarrow{e_i}$ such that:*

$$\forall \Sigma, \sigma.\ (\exists \Sigma_0.\ \Sigma_0, \Sigma, \sigma \models \mathit{Inv}(C))\underrightarrow{\ \Rightarrow\ \ \Sigma, \sigma}\models a) \quad \mathit{and} \quad \overrightarrow{a_{pre} \ast a \models_{\mathit{frm}} e_i} \quad \mathit{and}$$
$$\emptyset \vdash \{a_{pre} \ast a \ast \textbf{\textit{old}}(a_{pre} \ast a) \ast (\ast\, e_i = \textbf{\textit{old}}(e_i)\ )\}\ s_{body}\ \{\mathit{Inv}(C)\}$$

A program is valid if all of the message handlers of all actor classes are valid.

[8] Note that none of the Hoare Logic rules change this environment; the actor services included in Λ can be seen as hypotheses across the whole Hoare Logic proof.

The assertion a in the judgement above allows us information from the actor invariant which pertains to the current state. For example, permissions belonging to the actor invariant can be retained in a. Both this assertion and the message precondition are “duplicated” in the current and old states; our assertion semantics models these as independent states, but at the start of executing a message handler they should be known to be the same. The expressions e_i allow us to connect these two states with additional equalities, where relevant. The postcondition of the judgement requires that we can show that the actor invariant will hold across the entire message handler execution.

4.2 Obligation assertions

We allow our syntax of assertions used in Hoare triple pre- and postconditions to include *obligation assertions* of the form $\mathbf{obl}(o)$; intuitively these represent the requirement that we reach *some* program point at which a condition described by o is met. Here, o is either a *non-empty* response set R or the assertion $\mathit{localVariant}(\mathbf{this})$. In the former case, we require that the only variables mentioned in R are the parameters (including **this**) of the message handler being checked, and that *localVariant* assertions used in the where conditions of R are of the form $\mathit{localVariant}(\mathbf{this})$ (these relate to the local state of the actor; when proving a local service, we only have access to the local state of **this**).

We must define semantics for these novel assertions, including extending the standard notion of state to reflect obligations. The essential idea of our obligation semantics can be best understood by comparing with the semantics of permission assertions such as $\mathbf{acc}(e.f)$. In the case of permissions, such an assertion is true if it *under-approximates* the permissions actually held; we must hold *at least* the permissions required by the assertion. Obligation assertions have a dual semantics, they are true in a state if they *over-approximate* the actual obligations. This intuitively means that in a Hoare Logic proof it is allowed for permissions to be leaked but never fabricated ($\mathbf{acc}(e.f) \models^{\Lambda} \mathit{true}$ but the reverse does not hold); for obligations, the opposite is the case, and in particular, $A \models^{\Lambda} \mathit{true}$ does *not* hold in general, when A contains obligation assertions. Note that the assertion semantics does not reflect directly what the intended *meaning* of these obligations is; apart from forcing that they cannot be simply removed, they are treated as unknown assertions in the assertion semantics. Instead, their intended meaning is reflected in the Hoare Logic rules.

This design has an important outcome: obligation assertions included in the precondition of a Hoare triple cannot be removed using the rule of consequence. Instead, these can be removed only by the last three rules of Fig. 3, which specifically model the *discharge* of obligations. We can now show how to use obligation assertions to check local services against an implementation.

Definition 5 (Checking local services). *A handler for message m in class C* provides the local service $\forall \mathit{this}, \overrightarrow{X_i}.(\mathit{this}.m(\overrightarrow{X_i}) \leadsto R)$ under actor service environment Λ*, written $\Lambda, C, m \vdash \forall \mathit{this}, \overrightarrow{X_i}.(\mathit{this}.m(\overrightarrow{X_i}) \leadsto R)$, iff (where $a_{pre} = \mathit{pre}(m, \mathit{this}, \overrightarrow{X_i})$, and s_{body} is the body of the message handler for m in class C*

with formal parameters renamed to $\overrightarrow{X_i}$) there exist a one-state assertion a and expressions $\overrightarrow{e_i}$ such that:

$$\forall \Sigma, \sigma.\ (\exists \Sigma_0.\ \Sigma_0, \Sigma, \sigma \models \mathit{Inv}(C, \mathit{this}))\ \underrightarrow{\Rightarrow\ \Sigma, \sigma \models} a)\quad \mathit{and}\quad \overrightarrow{a_{pre} * a \models_{\mathit{frm}} e_i}$$
$$\mathit{and}\quad \Lambda \vdash \{a_{pre} * a * \textbf{\textit{old}}(a_{pre} * a) * (\ast\, e_i = \textbf{\textit{old}}(e_i)\ ) * \textbf{\textit{obl}}(R)\}\ s_{body}\ \{\mathit{true}\}$$

This definition is similar to Def. 4: we remove the requirement to check the actor invariant (we could keep this, but all message handlers must be checked to be valid in any case), allow for the possibility of a non-empty actor service environment, and, importantly, add the obligation assertion **obl**(R) to the precondition. The fact that *no* obligation assertions occur in the postcondition forces the message handler implementation to discharge these obligations before terminating.

4.3 Discharging obligations

Obligation assertions of the form **obl**(R), can be discharged if *one* of the alternatives described by the response pattern R can be shown to take place. In this section, we explain the last three rules of Fig. 3, which handle discharging obligations. To illustrate the rules, we consider the proof of local service (*QM3*) against the sols message handler in the QueryManager class.

In the case that R contains an empty response as an alternative, the rule (*emptyObl*) defines the criterion for the obligation to be discharged based on this alternative. The premises require that we can show that the where condition holds (recall that where conditions for empty responses may only constrain the "**old**" state); in this case, the **obl**(R) assertion need not be included in the postcondition. For example, in the (implicit) else branch at line 35, we can apply this rule, combining the fact $\neg$(**this**.next = **null** $\wedge$ **this**.user = **null**) known in this branch with **this**.next = **old**(**this**.next) $*$ (**this**.user = **old**(**this**.user)), from the precondition of the judgement (cf. Def. 5), to obtain the required where-clause.

The rule (*messageObl*) handles the similar (but more complex) case of discharging an obligation via a message send[9]. Conceptually, the first two lines of premises check that we have the message precondition, that the message send in the code matches that in the response pattern (when evaluating the receiver and parameter expressions in the current state), and that the where condition A_1 is framed by permissions in the current state. Intuitively, we would then just check that A_1 can also be shown to be true in this state. This is correct *except* in the case where A_1 includes *localVariant*(**this**) assertions. Whether a suitable local variant is established by this message handler cannot be determined at this program point; this can only be checked across the entire execution of the message handler. Our solution is to split the checking of A_1 into two parts: the requirements that the assertion makes independently of *localVariant* assertions (A_2, in the rule, which is then checked to hold), and the *condition* under which

[9] Note that rule (*messageNoObl*) can be applied instead of rule (*messageObl*), when one cannot or need not discharge an obligation via this message send statement.

A_1 requires a *localVariant*(**this**) assertion, which we use to prescribe a new obligation in the postcondition (B_2 is this condition)[10]. The last premise of the rule, which lets us potentially rewrite B_2 into a weaker condition B_3 (thus potentially requiring the resulting obligation more often) is necessary only in the case that B_2 depends on heap locations to which (exclusive) permission is given away (in a); this may force us to abstract the precise condition to one which can still be evaluated after the message send.

As an example of applying this rule, at line 40, we can use the (*messageObl*) rule; here, the corresponding A_1 assertion is simply *localVariant*(**this**), and so A_2 and B_2 are both *true*: this results in the assertion **obl**(*localVariant*(**this**)) in the postcondition. Note that at *this* program point, no suitable local variant has been established. By the end of the message handler, we will still have this obligation, but under the condition describing that we took the path through the message handler which reaches line 40.

By this point in the code (as claimed previously in Sec. 3.8), the expression $\textbf{this}.\mathsf{nrSolutions} - |\textbf{this}.\mathsf{store}|$ will have been decreased since the message handler began executing. We include in the actor invariant for QueryManager that the value of the store field can never decrease in size, while the nrSolutions field is immutable (once initialised): we then know that the actor invariant guarantees that this expression will never increase in value across the execution of subsequent message handlers. These conditions make up the premises of the rule (*localVarObl*), which can be used to discharge this obligation in the proof of this message handler.

4.4 Overall proof strategy: iterated derivation of actor services

In order to prove local services which include actor services in their where-clauses, the rules of Fig. 3 are already sufficient. However, at the point of applying the rule (*messageObl*) explained above, it will be necessary to discharge a premise that shows that the where-clause holds at this point. When actor services occur in this where-clause, this will be possible only if the Hoare triple precondition already includes actor services, or we are able to derive them from it. In either case, it will be possible to prove any actor services *during* a Hoare Logic proof only if the actor service environment Λ is not empty. Including a non-empty actor service environment Λ makes the justification of a new local service hypothetical. We must ensure that this yields a well-founded derivation of the eventual response property; it would not be acceptable to prove a local service by first assuming it in Λ and then deriving it according to Def. 5.

We can support hierarchical derivation of actor services (new services are derived using only the results of previous proofs), as follows: firstly, some local services can be derived with an empty actor service environment. Any actor service without nested actor services in its where-clauses will (if derivable at all) not require any assumed actor services. In our running example, the three local

[10] This splitting can be achieved syntactically; we abstract its definition here with the function *splitLocalVariant*.

services (*QW*), (*QM2*) and (*QM3*) fall into this category. Then, by *assuming* these local services, we are able to derive e.g. (*QM*), whose derivation requires the arguments presented throughout Sec. 3 in order to justify the nested actor service. If local services are built up in this hierarchical fashion, the justification of the corresponding response properties is guaranteed to be well-founded.

Definition 6 (Iterative derivation of actor services). *An actor service environment Λ can be* iteratively derived *(for a given program), if there exists* $n \geq 0$ *and there exist actor service environments $\Lambda'_1, \Lambda_1, \Lambda'_2, \Lambda_2, \ldots, \Lambda'_n, \Lambda_n$ such that (taking $\Lambda_0 = \emptyset$), we have $\Lambda_n = \Lambda$ and for all $0 \leq i < n$:*
(1) Each $\forall \overrightarrow{X_i}.(T \leadsto R) \in \Lambda'_{i+1}$ is a local service, and $\Lambda_i, C, m \vdash \forall \overrightarrow{X_i}.T \leadsto R$ (where C, m are the class and message name of the trigger message T).
(2) For each $\forall \overrightarrow{X_i}.(T \leadsto R) \in \Lambda_{i+1}$, we have true $\models^{\Lambda'_{i+1}} \forall \overrightarrow{X_i}.T \leadsto R$.

This definition allows us to alternate between deriving new local services (Λ'_{i+1}) based on previously derived actor services, or deriving new actor services (Λ_{i+1}) from local services. The latter step can be useful for information hiding reasons: if we wish to present an actor service environment as a specification for part of the program, then presenting only local services may not be suitable: because a local service must always expose a response message which is sent directly by the message handler for its trigger message, this might expose details (of intermediate actors, messages and field names) that we do not wish to. Being able to rewrite these local services into arbitrary actor services allows us to avoid this (in our running example, this wasn't necessary, since the local service (*QM*) was derivable without exposing internal details).

4.5 Soundness

The proof technique of Def. 6 also lends itself to proving soundness of our actor service reasoning. While a formal operational semantics and soundness argument are beyond the scope of this paper, we summarise the essential points here.

Firstly, we consider only well-typed programs which are valid (Def. 4). Operationally, we consider null dereferences and data races as runtime errors. We define a notion of valid runtime state, which includes the requirement that a suitable partioning of the heap into *owned* regions and an *immutable* region must exist. As the program executes, this partitioning will change, but locations in the *immutable* region will remain so. Note that this notion of ownership is an artifact of the argument but not a feature of the operational semantics itself (which does not track permissions). Based on this idea, we can show that a valid program will never get stuck or encounter runtime errors.

To tackle the soundness of actor services, we need to relate these assertions to the intended temporal (response) property of runtime traces which they notionally represent. We can show two key results. Firstly, we can show that for any *local service* $\forall \overrightarrow{X_i}.(T \leadsto R)$ such that $\Lambda, C, m \vdash \forall \overrightarrow{X_i}.(T \leadsto R)$, any execution of the message handler for m in C will either not terminate, or will eventually reach a state in which one of the response messages is sent (or none, if an empty response pattern is included), and the corresponding where-clause will be *derivable* at this point, possibly using the actor services assumed in Λ. This result can be shown in a simplified operational semantics in which we only consider the local execution of a single actor executing the appropriate message handler. The result can be made a true liveness property if one chooses to also prove termination of each message handler, which we regard as an orthogonal problem.

Secondly, we can show tha for any actor service $\forall\overrightarrow{X_i}.(T \rightsquigarrow R)$ derivable in a program state under an actor service environment Λ, if we *assume* that all actor services in Λ describe valid response properties of the runtime traces of the program, then the actor service $\forall\overrightarrow{X_i}.(T \rightsquigarrow R)$ will also do so. This requires an induction over the derivation (according to the rules of Fig. 2) of $\forall\overrightarrow{X_i}.(T \rightsquigarrow R)$, and requires that sent messages are always eventually delivered (not necessarily in order), selection of a new message to execute is weakly fair, and all actors continue to respond (i.e., will always eventually receive another message, if any are waiting); in particular, that no actor executes a message handler forever.

As a corollary of these two results, we obtain that, in a valid program, any actor service which can be iteratively derived (Def. 6) will describe a response property true for the runtime traces of the program, under the same assumptions.

5 Related Work

With respect to the case study of Arts and Dam [2], we provide a simple proof of the same response property, using the actor services, permissions and actor invariants provided by our technique. Poetzsch-Heffter *et al.* [32] argue the need for actor reasoning techniques supporting compositionality. Kurnia and Poetzsch-Heffter [22] present such a compositional technique based on trace-based assertions. However, behaviours guaranteed by actors can only be summarised in a hierarchical fashion, according to a fixed topology [34], and call-backs in sequences of messages (such as in the ring in our example) are not permitted.

A number of other techniques base reasoning around invariants over *histories* of message-events (e.g. [13, 1, 12]). These approaches can express intricate properties over many successive events. The techniques involve one verification at the level of individual actors (via invariants), followed by a composition phase to derive a system-wide invariant. These safety properties cannot guarantee that certain response events will occur (after perhaps unrelated actions by the same actor). Individual actors can be verified in isolation, but it is not possible to summarise behaviour of parts of a program such that the summaries can themselves be composed later. Feng [16] makes a similar argument for hierarchical compositionality in the context of reasoning about concurrency and shared data.

Some language designs provide guarantees about actor-like programs *by design*, via reduction to model-checking [11, 10], custom support in proof assistants [33], or via high-level program descriptions from which code can be safely generated [8]. These works (as well as those based on temporal logics) require reasoning in terms of the whole program and/or do not handle liveness properties.

Multiple type systems have been proposed for guaranteeing properties such as copyless messaging (ownership transfer) and immutability in actor-like programs, e.g. [17, 36, 41, 6, 7]. These type annotations could be mapped onto permissions, to combine these systems with our technique for proving response properties. Some techniques integrate protocol verification [38, 4, 39, 15], which makes reasoning more precise and able to address other safety properties.

Extensive work has been carried out on protocol verification for message-passing programs, using (multiparty) session types [20, 21, 9, 3]; such work typically does not address liveness or compositional reasoning. Such protocol reasoning could, however, complement our proof technique (see Sec. 6). Padovani *et al.* [29] address liveness at the protocol level; this does not guarantee that the underlying code will continue to produce messages, and does not support compositionality. Lange and Tuosto [23] show how to synthesise global descriptions of a system from local ones; this is closer in spirit to our compositional reasoning, but does not support functional specifications or liveness.

The "causal obligations" of Helm *et al.* [18] are a specification construct similar to simple local services. No proof system was defined for this construct.

6 Conclusions and Future Work

We have introduced a new modular verification technique for programs which communicate via asynchronous messaging. Our proof technique is compatible with permission-based logics; in particular, it is straightforward to adapt the assertion logic (including the where-clauses of our actor services) to incorporate standard features of these logics such as abstract predicates [30]. Our semantics for actor service assertions is largely orthogonal to the particular logic used for functional specification, provided that two-state properties can be expressed.

A natural extension is to generalise the form of actor services to express response properties with more than one response message and more than one trigger message. The former extension is straightforward, but the latter requires additional book-keeping in the reasoning, in order to represent the case that some but not all of the trigger messages have been received; we leave this extension for future work, along with a complete formalisation and soundness proof.

Our current proof technique assumes that actors must, by default, be always ready to receive any message permitted by their type. It would be interesting to combine our actor service reasoning with protocol verification techniques (such as session types and typestate reasoning), in order to make our technique more expressive and to support explicit de-allocation of actors.

Acknowledgements. We are very grateful to Sophia Drossopoulou, Sylvan Clebsch, Juliana Franco, Tim Wood, Susan Eisenbach, Malte Schwerhoff, Ernie Cohen, John Boyland, Arnd Poetzsch-Heffter, Ludovic Henrio, Pietro Ferrara and Reuben Rowe for important technical discussions and encouragement. We thank Mariangiola Dezani and Dilian Gurov for pointers to related work, and the ESOP reviewers for many helpful suggestions which have improved the paper.

References

1. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 77(12):1289–1309, Oct. 2012.
2. T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *FM 1999*, volume 1708 of *LNCS*, pages 682–700, 1999.
3. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010 - Concurrency Theory*, volume 6269 of *LNCS*, pages 162–176. Springer Berlin Heidelberg, 2010.
4. V. Bono, C. Messa, and L. Padovani. Typing copyless message passing. In *TOPLAS*, volume 6602 of *LNCS*, pages 57–76, 2011.
5. J. Boyland. Checking interference with fractional permissions. In *SAS 2003*, volume 2694 of *LNCS*, pages 55–72, 2003.
6. D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *TOPLAS*, volume 5356 of *LNCS*, pages 139–154, 2008.
7. S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *AGERE! 2015*, pages 1–12, 2015.
8. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies. In *COORDINATION 2015*, volume 9037 of *LNCS*, pages 67–82, 2015.
9. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. *SIGPLAN Not.*, 46(1):435–446, Jan. 2011.
10. A. Desai, P. Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In *OOPSLA 2014*, pages 709–725, 2014.
11. A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. *SIGPLAN Not.*, 48(6):321–332, June 2013.
12. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *The Journal of Logic and Algebraic Programming*, 81(3):227 – 256, 2012. Proc. of NWPT 2010.
13. J. Dovland, E. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *SwSTE 2005*, pages 141–150, Feb 2005.
14. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP 2008*, volume 5142 of *LNCS*, pages 412–437, 2008.
15. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *In EuroSys*, pages 177–190. ACM Press, 2006.
16. X. Feng. Local rely-guarantee reasoning. In *POPL 2009*, pages 315–327, 2009.
17. P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010*, pages 354–378, 2010.
18. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *OOPSLA/ECOOP 1990*, pages 169–180, 1990.
19. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI 1973*, pages 235–245, 1973.
20. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP 1998*, pages 122–138, 1998.

21. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, Jan. 2008.
22. I. W. Kurnia and A. Poetzsch-Heffter. A relational trace logic for simple hierarchical actor-based component systems. In *AGERE! 2012*, pages 47–58, Oct. 2012.
23. J. Lange and E. Tuosto. Synthesising choreographies from local session types. In *CONCUR 2012*, pages 225–239, 2012.
24. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP 2009*, volume 5502 of *LNCS*, pages 378–393, 2009.
25. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.
26. H. Nilsson. Method and apparatus for evaluating a data processing request performed by distributed processes. Patent Filed 1998, issued August 2003: `http://patents.justia.com/patent/6604122`.
27. M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the Scala programming language. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2004.
28. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL 2001*, pages 1–19, 2001.
29. L. Padovani, V. T. Vasconcelos, and H. T. Vieira. Typing liveness in multiparty communicating systems. In *COORDINATION 2014*, volume 8459 of *LNCS*, pages 147–162, 2014.
30. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL 2005*, pages 247–258. ACM Press, 2005.
31. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.
32. A. Poetzsch-Heffter, I. W. Kurnia, and C. Feller. Verification of actor systems needs specification techniques for strong causality and hierarchical reasoning. In *FoVeOOS 2011*, pages 289–305. Technische Universität Karlsruhe, October 2011.
33. D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner. Automating formal proofs for reactive systems. *SIGPLAN Not.*, 49(6):452–462, June 2014.
34. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP 2010*, LNCS, pages 275–299, June 2010.
35. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 148–172, July 2009.
36. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP 2008*, volume 5142 of *LNCS*, pages 104–128, 2008.
37. A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *ECOOP 2013*, volume 7920 of *LNCS*, pages 129–153, 2013.
38. J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In *TOPLAS*, volume 5904 of *LNCS*, pages 194–209, 2009.
39. J. Villard, É. Lozes, and C. Calcagno. Tracking heaps that hop with heap-hop. In *TACAS 2010*, volume 6015 of *LNCS*, pages 275–279, 2010.
40. R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 2nd edition, 1996.
41. Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic Java. *SIGPLAN Not.*, 45(10):598–617, Oct. 2010.