Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Modular Verification of Response Properties in Protocol-Based Actor Programs

Master's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

Author:
Gaurav Parthasarathy

Supervisors:
Dr. Alexander J. Summers
Prof. Dr. Peter Müller

May 12, 2018

Acknowledgements

I would like to thank my primary supervisor Dr. Alexander J. Summers, without whom many of the ideas developed in this thesis would not have come to fruition. Alex, thanks for the many interesting discussions which always made me think about challenging problems in a much more elegant way and for your invaluable guidance.

I also would like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this interesting research project.

Finally, I would like to thank my parents for supporting me throughout my studies.

Contents

1 Introduction

Concurrent programs have become ubiquitous. In practice, such programs must deal with a large number of processes running concurrently. One needs some way for the processes to interact safely. Using shared memory locations requires careful synchronization between the processes which quickly leads to issues when scaling to large programs.

One approach to avoid this issue is to restrict communication between processes to asynchronous messages. The actor model [10, 2] is a programming paradigm that follows this asynchronous messaging approach, where processes are independent units called *actors*. Each actor has its own local state and a mailbox which holds all messages that the actor has not yet reacted to. As a reaction to a message an actor may spawn other actors, send messages to other actors and change how it reacts to future messages.

The actor model facilitates the development of large-scale, robust concurrent programs. There are various programming languages and frameworks that can be used to write concurrent programs using this model. Examples include Akka [22], Erlang [3], Kilim [19] and Pony[1].

While the actor model provides many benefits, it is not always easy to reason about actors, which we illustrate by considering the simple partial actor program written using Akka in Figure 1. The figure shows implementations for the client, manager and worker actor types, for which there may be an arbitrary number of actor instances. The manager's initial behaviour, describing how it reacts to received messages, is given by the acceptQuery method. While the manager reacts to messages according to this initial behaviour, only Query messages are accepted without an exception being thrown.

The Query message specifies a client interested in an answer and a value n specifying the query. Once the manager removes a Query message from its mailbox (at most one message is handled at a time), the manager sends a Compute message to some worker actor (line 14) (assuming the manager behaves according to its initial behaviour). After this, the manager changes its behaviour to another one, in which it only accepts Result messages (line 15) without throwing exceptions. The worker computes $f(n)$ and sends it back via a Result message to the manager, which sends the received result to the client and moves back to its initial behaviour.

It is essential for the manager to define two such different behaviours and to not react to Query and Result messages the same way at all times. When the manager is initially waiting for a Query message and it receives a Result message, then the manager does not know for whom this Result message is intended, hence it cannot send it to a client as on line 21. It is also not

```
class Client extends Actor {
    def receive = {
        case Solution(sol) => ...
        case _ => ...
    }
}

class Manager extends Actor {
    def receive = acceptQuery

    def acceptQuery : Receive = {
        case Query(client, n) =>
            val worker = context.actorOf(Props[Worker])
            worker ! Compute(self, n)
            context.become(waitForResult(client))
        case _ => throw new Exception(”unexpected message”);
    }

    def waitForResult(client: ActorRef) : Receive = {
        case Result(res) =>
            client ! Solution(res)
            context.become(acceptQuery)
        case _ => throw new Exception(”unexpected message”);
    }
}

class Worker extends Actor {
    def receive = {
        case Compute(manager, n) =>
            manager ! Result(f(n))
        case _ => ...
    }
}
```

Figure 1: Partial Akka actor program written in Scala showing implementations for three different types of actors. The manager actor changes its behaviour using the built-in become *primitive. The initial behaviour of each actor is given by the* receive *method. Message sending is expressed using the syntax* recipient ! message*. Actor creation is expressed using the syntax* context.actorOf(Props[ActorType])*.* self *holds the reference to the current actor.*

acceptable for the manager to send the result to some client (for example, to the previously handled client), because this client may have just sent a Query message to the manager, which the manager has not yet processed. The reason is if the manager sends the result to this client, then this client will assume that it received the result corresponding to the Query message that it sent, which is incorrect.

When the manager has sent a Compute message to a worker and is waiting for the corresponding Result message, then the manager cannot react to Query messages the same way as according to its initial behaviour, because in its implementation it only keeps track of a single client. In this implementation the manager just throws an exception, if such a Query message is received. Another solution would be to store the Query message in its local state and to process it once the manager moves back to its initial behaviour. However, this also requires a different way of reacting to the Query message than in the initial behaviour.

So, the manager is following a fixed protocol, where it reacts to messages differently dependent on how far the protocol has progressed. We call such actor programs *protocol-based actor programs*. The coordination required to implement actors that follow protocols, as can already be seen in the simple example, does not very directly reflect the actual protocol itself (especially if the protocol is complex). This makes it hard to reason about such protocol-based actor programs. Therefore there is a need to develop techniques which allow reasoning about such programs.

Consider an actor that sends a Query message to the manager. In general, it is not clear what will happen next. When the manager receives this message, while being in a state in which it throws exceptions upon receiving Query messages, then we do not get the desired properties. We would like to be sure that when this message is sent to the manager with input n, then eventually the client specified in the query message will receive $f(n)$. Without such a guarantee, the actor program has no value. There are various challenges when attempting to verify this desired response property in the program (which may contain other actor types not shown):

1. It must be ensured that if a message is sent to the manager, then the manager is guaranteed to be in a state, where it will accept this message without throwing an exception, until the manager removes the message from its mailbox. For example, the manager should not receive a Query message while it is waiting for a Result message.

2. One must be able to relate the client value on line 21 with the client value provided by the Query message on line 12.

3. The Result message received by the manager should contain the correct value $f(n)$. One needs to make sure that nobody sends some Result containing a wrong value to the manager.

In the presented example only the manager is following a non-trivial protocol. As actor programs get more complex and multiple actors follow non-trivial protocols more challenges arise. Verification techniques which require knowledge about the complete program do not scale since a change to part of the program invalidates the complete proof, also they are not applicable in cases where the complete program is not known. Therefore a modular verification approach is desirable, where it is possible to verify actors independently and where adding verified actors to an already existing system does not violate the already derived properties. Furthermore, it should be possible to derive response properties modularly. This means one should be able to compose response properties expressing behaviours of different parts of the program to derive a response property summarizing a larger part of the program. Additionally, whenever one part of a program changes, one should still be able to reuse a derived response property for an unaffected part of the program.

Achieving such modularity in protocol-based programs is challenging for various reasons. The response properties, as the one presented, only mention the initial request and the final response, abstracting over what happens in between. However, a response may advance the protocols of multiple actors which are not mentioned in the request or in the response. When composing such a response property with another response property, one might need to know how those protocols advanced. Furthermore, one also needs to relate the local states of the actors in the different protocol stages to the request and the response parameters to be able to prove how the final response value relates to the initial request value.

We propose a modular verification technique for proving response properties for an interesting class of protocol-based actor programs (such as the one given in Figure 1) by extending the actor services program logic [20]. We achieve all the modularity features described.

Outline. In Section 2 we provide the necessary background for the actor services logic. We show the limitations of the logic with respect to protocol-based actor programs in Section 3. In Section 4 we introduce support for protocols in the actor services logic using a notion of a session. We extend the introduced protocols in Section 5 to also include fork-join patterns. We discuss related work in Section 6. Finally, we conclude and provide directions for future work in Section 7.

```
actor Client {
    int val;
    Client(int v) { this.val := v }
    handler sol(int res) {...}
}
actor Master {
    Client c; Worker worker;
    handler getsol(Client client) {
        this.c := client;
        Worker w1 := spawn Worker();
        this.worker := w1;
        w1.sendsol(client, client.val);
    }
}
actor Worker {
    handler sendsol(Client client, int n) {
        client.sol(f(n));
    }
}
```

Figure 2: A simple actor program where the master actor delegates work to a worker actor that it spawns.

2 Background: The actor services program logic

The actor services program logic [20] is a logic to verify response properties in actor programs modularly. In this section we first introduce the programming language used to express the actor programs and compare it to existing actor languages. Next, we introduce the parts of the logic that are relevant for the subsequent sections. Finally, we present the logic's limitations that we address in later sections.

2.1 The actor programming language ActorPL

The actor services program logic works with a simple Java-like programming language with which actor programs can be expressed. We call the programming language *ActorPL* and we present it by considering the actor program given in Figure 2. Each actor is an instance of an *actor class*, which is declared with the **actor** keyword, containing the actor's implementation. There may be arbitrarily many instances of a single class. An actor class defines for each message that can be received by the actor, a message handler declared with the **handler** keyword. The message handler signature specifies the arguments of the message as well as their corresponding types. For example, in Figure 2 the client actor may only receive sol messages and these messages must have one integer argument. **this** is a reference to the current actor instance.

The semantics of ActorPL reflect the actor model. Each actor has an implicit message queue, which we refer to as the actor's *mailbox*, containing all the messages that it has received but has not yet reacted to yet. An actor can send a message to another actor by addressing the corresponding message handler. This operation is non-blocking and the effect is that the message is sent to the actor's mailbox. The message arguments must conform to the message handler signature. In the example the worker sends a sol message to the client using the expression client.sol(f(n)).

Each actor class A contains a single constructor $A(\ldots)$ with potential arguments[1]. The client has the constructor Client(**int** v). It is not permitted to send messages inside a constructor. If no constructor is specified (as in the manager and worker), then this is the same as having an empty constructor with no arguments. An actor can be spawned using the **spawn** keyword and providing values for the arguments of the constructor. This can be seen in the getsol message handler of the master, where a worker is spawned. The constructor is invoked synchronously when the actor is spawned. Once an actor is spawned it enters an implicit loop. In each iteration the actor removes a message from its mailbox and the body of the corresponding message handler is executed, after which the next iteration starts. Therefore in a single actor there is at most one message being handled at any given time. If there is no message in the mailbox then the actor waits until a message arrives. It is not assumed that messages arrive in order. However, it is assumed that if a message is sent to a mailbox then it will eventually reach this mailbox and if the actor is guaranteed to keep removing messages from its mailbox then eventually the message will be removed from the mailbox. We refer to this as the *weakly-fair message receive assumption*. Furthermore, it is assumed that messages are never duplicated in transit.

In ActorPL there is a heap and actor classes can define fields which are stored in the heap (see the val field in the client actor). All the fields of the different actor instances are modelled as being part of the same heap. The logic supports reasoning about disjoint heap locations by having a notion of ownership for the heap locations that is associated with the actors. To do

[1]Constructors are not supported in [20].

```
actor A {
    handler start(B b, int i) {
        b.init(i);
        b.get(this);
    }
}
```

Figure 3: Even though the init message is sent before the get message actor b might receive the get message first.

this reasoning one needs to provide specification annotations for the actor classes which is something that ActorPL supports as well. We present this in Section 2.5.

2.2 Differences to existing actor implementations

There are many different programming languages and frameworks that support some form of the actor model. While the main idea in these implementations is the same, there are important differences. Next, we compare ActorPL with some well-known actor languages.

2.2.1 Message ordering

As mentioned in Section 2.1 there are no assumptions made on the order in which messages arrive in a mailbox. For example, consider the actor A in Figure 3. Actor A first sends the init message to actor b followed by the get message. Since no assumptions are made on message ordering b might receive get before init. In certain existing actor implementations such as Akka and Erlang it is guaranteed that init is received before get. Since ActorPL makes less assumptions with respect to message ordering this means that verified properties also hold for programs with stronger message ordering guarantees.

2.2.2 Behaviour change

In Section 1 we introduced the Akka program given in Figure 1. In this program the manager actor changes its behaviour using the become primitive and whenever it receives a message that it does not expect according to its current behaviour an exception is thrown. Furthermore, the client value is carried along from the Query message to the next behaviour. A possible ActorPL encoding of the Akka actor program is given in Figure 4. We introduce a global method fail() which encodes that an exception is thrown.

```
actor Client {
    handler sol(int res) { ... }
}

actor Manager {
    Client client;
    State state;

    Manager() { this.state := ACCEPTQUERY }

    handler query(Client client, int n) {
        if(this.state == ACCEPTQUERY) {
            this.client := client;
            this.state := WAITFORRESULT;
            Worker worker := spawn Worker();
            worker.compute(this, n);
        } else {
            fail();
        }
    }

    handler result(int res) {
        if(this.state == WAITFORRESULT) {
            this.state := ACCEPTQUERY;
            this.client.sol(res);
        } else {
            fail();
        }
    }
}

actor Worker {
    handler compute(Manager m, int n) {
        m.result(f(n));
    }
}
```

Figure 4: ActorPL encoding of Akka actor program given in Figure 1.

In our semantics one does not need to show the absence of such fail() methods. We use them to explicitly indicate that an actor reacts differently to a certain message compared to the expected case. For the encoding the idea is to introduce a state field which indicates the current behaviour of the manager. Furthermore, the actor's heap is used to keep track of the client value. Of course one must ensure that only the manager can change its own state and client fields, otherwise the semantics of the encoding would not reflect the semantics of the original program. We will see later how to ensure this using specification annotations.

Note that one difference to the original Akka actor program in Figure 1 is that the actor provided in the query message is known to be a client actor. This difference stems from the fact that in Akka actor references corresponding to different actor types are not distinguished. However, since the manager's behaviour does not depend on the type of the actor received over the query message, this is just a technical difference.

2.2.3 Selective receive

Akka actors need to specify in each state all the messages that they accept. If a message is removed from the mailbox in a state where it is not accepted, this message is classified as an unhandled message and an exception is thrown. This can be encoded by ActorPL, as shown in Section 2.2.2, by storing the current state of the actor in its local heap and checking in each message handler if the actor is in a state where the message is accepted. Erlang supports a different approach. In Erlang it is possible to specify which type of messages are removed from the mailbox in any given state. This means it is guaranteed that only the specified messages will be removed from the mailbox and all other messages remain in the mailbox. If a message is in the mailbox that currently is not accepted, it is not regarded as a failure: the message may be removed in a future state where it is accepted. This feature is known as *selective receive*.

One can encode selective receive in ActorPL with some semantic differences. The idea is that one again uses the local heap in the actor to identify the current behaviour and to track information passed between behaviours. Whenever a message handler is invoked, one can use a conditional statement to figure out if the message that was just removed from the mailbox is accepted. If the message is not accepted, then one can send the message back to oneself which puts it back into the mailbox. This approach is similar to how Akka simulates *selective receive* using *stashing*. The semantic difference is that messages that are not accepted are explicitly removed (and reinserted).

2.3 Introducing actor services

The fundamental ingredients of the actor services logic are actor services. Actor services are assertions that express response properties. We introduce them by considering the actor program given in Figure 2. For example, the actor service

$$\underline{W}.\mathsf{sendsol}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n)) \tag{2.1}$$

holds in any program state where we use the notation that a variable is universally quantified if its first occurence in the actor service is underlined [2]. It states the following: for all sendsol messages received by worker W in the future (with respect to the current state) with client C and integer n as arguments, it is guaranteed that eventually client C will receive a sol message with argument $f(n)$. By "W receives" we mean W removes the message from its mailbox. It can be easily seen by inspecting the implementation that this property holds in any program state.

The left hand side of an actor service (in this case $\underline{W}.\mathsf{sendsol}(\underline{C}, \underline{n})$) is called the *trigger message* and the right hand side of an actor service is called the *response pattern* describing the different possible *response messages* (in this case only one possible response message is specified, namely $C.\mathsf{sol}(f(n))$). Generally speaking, if an actor service holds in the current state then this means that for all future trigger messages eventually one of the response messages will be sent. It is important to note that the response message is guaranteed to be sent strictly after the trigger message is received.

Note that actor services do not specify who sends the response message. If an actor service holds in the current state then it holds in every future state as well. In summary one can say that actor services express two different things:

1. They express a *liveness property* stating that some response message will certainly occur in the future if a trigger message is received.

2. They express a *functional property* which relates the values in the response state with the values in the trigger state.

2.4 Actor services dependent on the program state

The actor service (2.1) holds in all program states. In particular, it expresses a response property in terms of arbitrary workers. It is possible to derive an actor service from (2.1) in terms of a specific worker. For example, in the program state corresponding to line 11 in Figure 2 the local variable w1 points

[2]Formally one would write $\forall W, C, n.\ W.\mathsf{sendsol}(C, n) \rightsquigarrow C.\mathsf{sol}(f(n))$

to a worker actor. One can instantiate the universally quantified worker variable in (2.1) to get the actor service $\mathsf{w1.sendsol}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n))$. This actor service depends on the program state since it only talks about the trigger messages received by worker w1. It can be derived in those program states where w1 is a local variable pointing to a worker actor; the program state corresponding to line 11 is one such state.

It is also possible to instantiate the actor service with heap-dependent expressions, but to understand when this is fine and to present the meaning of the resulting actor services one must understand the differences between expressions in the trigger message and response messages. All the heap-dependent expressions in the trigger message are evaluated in the current state (i.e. in the state in which the actor service holds). All the heap-dependent expressions in the response message are evaluated in the corresponding response state (i.e. in the state in which the response message is sent). This distinction might seem strange but it enables the derivation of actor services where certain fields are only set appropriately right before the response is sent. For example, consider the following actor service that can be derived in an arbitrary program state

$$\underline{M}.\mathsf{getsol}(\underline{C}) \rightsquigarrow M.\mathsf{worker.sendsol}(C, C.val)$$

This actor service states that whenever a master M receives a getsol message in the future then eventually a sendsol message will be sent to the worker stored in the worker field of M *at the time when the message is sent.* The worker field is only set once the master receives the message, so if M.worker were evaluated in the current state then this actor service would have a different meaning and it would not hold in the program.

This interpretation of heap-dependent expressions requires caution when instantiating actor services with heap-dependent expressions. One may only instantiate a quantified variable with a heap-dependent expression if either the variable only appears in the trigger message (since there the expression is evaluated with respect to the current state) or if one can show that the expression evaluates to the same value in all future states (using immutability predicates which we introduce later). Since the worker variable in actor service (2.1) only appears on the left hand side of the actor service we can soundly instantiate it with **this**.worker to get

$$\textbf{this}.\mathsf{worker.sendsol}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n))$$

2.5 Specification annotations, permissions and immutability

Actor programs often make implicit assumptions which, if violated, lead to undesired program behaviour. For example, in Figure 4 an implicit assumption is that only the manager can modify its own `client` field. If this assumption were violated, then the message sent on line 25 would go to a potentially unknown client, which does not reflect the intended behaviour of the program. Verification of such actor programs requires making these assumptions explicit. In the actor services program logic there are three different types of specifications for actor programs: *message preconditions*, *actor invariants* and *constructor postconditions*.

Message preconditions are assertions that can be provided as a specification for a message handler. Any actor that sends a message m must satisfy the message precondition and in the message handler implementation one may assume this precondition. Hence message preconditions restrict the behaviours of the program, since they restrict when a message can be sent. This means when verifying an ActorPL program one must make sure that the message preconditions allow all the intended program behaviours.

For each actor class one can specify an actor invariant assertion. The actor invariant must be guaranteed to hold when an actor is spawned, i.e. at the end of the corresponding constructor. Furthermore, it must be guaranteed to be invariant with respect to all message handlers in the actor. This means the actor invariant may be assumed at the beginning of the message handler but must be ensured at again at the end of the message handler.

One may also specify a constructor postcondition[3] for the constructor of an actor. This must be guaranteed (along with the actor invariant) at the end of the constructor and may be assumed by the entity spawning the actor.

In these specifications the logic supports reasoning about the heap using permission-based reasoning in a similar fashion to implicit dynamic frames [18]. The *accessibility predicate* assertion **acc**$(x.f)$ represents *exclusive permission* to heap location $x.f$. If this assertion holds, then no one else has access to this heap location and therefore it is safe to write to and read from the location. For example, we may put **acc**$(this.worker)$ into the master's actor invariant in Figure 2 to ensure that the master may always write to $this.worker$ safely.

The *immutability predicate* assertion **immut**$(x.f)$ represents the *immutable permission* to heap location $x.f$. It states that $x.f$ is guaranteed to have the same value from the current program state onwards (i.e. in all future states). Hence, it is safe to read from a heap location if such a predicate is held. The

[3]Since [20] do not use constructors they also do not have any constructor postconditions.

logic permits transforming an exclusive permission $\textbf{acc}(x.f)$ into immutable permission $\textbf{immut}(x.f)$. This transformation is irreversible.

In specifications one may use the *separating conjunction* (see [16]) $A * B$ where A and B are assertions. $A * B$ holds if

- For each immutability predicate specified in A or B it is guaranteed that the corresponding heap location is immutable.
- The sum of the exclusive permissions specified in A and B is held.
- A and B hold.

This means that $\textbf{acc}(x.f) * \textbf{acc}(x.f) \models \textit{false}$ since one cannot own the exclusive permission twice and also $\textbf{acc}(x.f) * \textbf{immut}(x.f) \models \textit{false}$ since immutability implies that no one has exclusive permission.

Actor invariants and message preconditions must be *self-framing*. An expression is framed by an assertion, if permissions to all heap-dependent subexpressions are held in the assertion. An assertion A is self-framing if it frames all expressions occurring in A. This guarantees that one may talk about all the mentioned heap locations safely. For example, $\textbf{acc}(x.f) * x.f$ is self-framing, since it holds exclusive permission to $x.f$. $\textbf{immut}(x.f) * x.f$ is self-framing as well. The expression $x.f$ is framed by assertion $\textbf{acc}(x.f)$.

One may transfer permission to heap locations over messages by specifying the permissions in the corresponding precondition. For example, specifying $\textbf{acc}(x.f) * \textbf{acc}(y.f)$ in the precondition means that the sender must give up the permissions to $x.f$ and $y.f$.

2.6 where-clauses

Until now we have only considered actor services that describe a liveness property and a basic functional property which relates the arguments in the trigger message with the arguments in the response message. In general, one can also relate part of the program state in the *trigger state* (i.e. the state when the trigger message is received) with part of the program state in the *response state* (i.e. the state when the response message is received) using *where-clauses*.

An actor service is of the form $\forall \overrightarrow{X_j}.(e.m(\overrightarrow{e_i}) \rightsquigarrow R)$ where R is a response pattern. A response pattern consists of a set of response messages (denoted $r_1 \mid r_2 \mid ... \mid r_n$). The meaning is that whenever the trigger message is received in a future state, one of the response messages will be sent. Until now we have only considered response messages of the form $e'.m(\overrightarrow{e_i'})$, but the logic permits response messages of the form

$$e'.m(\overrightarrow{e'_i})\ \mathit{where}\ A$$

where A is the where-clause of the response message. A is a *two-state assertions* which can contain *old expressions* of the form $\textbf{old}(e)$ where e is evaluated in the trigger state. All expressions that are not old expressions are evaluated in the response state. Furthermore, A cannot contain accessibility predicates.

The meaning of an actor service with such a response message r is that if r is sent, then A will be guaranteed to hold at that point. For example, consider the following actor service

$$\begin{gathered}\underline{M}.\mathsf{getsol}(\underline{C}) \rightsquigarrow \exists n.\ M.\mathit{worker}.\mathsf{sendsol}(C, n)\\ \mathit{where}\ (\textbf{old}(C.\mathit{val}) = C.\mathit{val}) * n = C.\mathit{val}\end{gathered}$$

where M denotes a manager actor from Figure 2. This actor services states that whenever M receives a getsol message, then eventually a sendsol message will be sent to the manager's worker. Additionally, we have that the val field of the client C when the getsol message is received is the same as when the response is sent and the value equals the input that is provided sendsol message.

Where-clauses may only mention heap-dependent expressions that are framed. This means that these expressions are either framed by immutability predicates in the where-clause itself or by the corresponding preconditions. In the given actor service since there are no immutability predicates in the where-clause, $\textbf{old}(C.\mathit{val})$ must be framed by the precondition of $M.\mathsf{getsol}(C)$ and C.val must be framed by the precondition of $M.\mathsf{sendsol}$.

An intuition why immutability predicates are permitted and accessibility predicates are not permitted in such where-clauses is that immutability predicates essentially express knowledge about a heap-location that remain true in every future state and hence such an immutability predicate may be freely duplicated. Accessibility predicates, however, contain *unique* ownership that cannot be duplicated, making their meaning in a where-clause unclear.

If the framing conditions hold, then we say that the where-clause is *framed*. The motivation for these conditions will become clear in Section 2.7.

2.7 Proving actor services

In the previous sections we only talked about what the meaning of actor services are but not how they can be proved. The actor services logic differentiates between *local actor services* and *non-local actor services*. Local actor services are actor services that can be proved by just considering the implementation of the trigger message handler. For example, actor service (2.1) is a local actor service. In the worker's sendsol message handler in Figure 2 the client's sol response message is sent. Local actor services hold in any program state.

The logic defines a Hoare logic to prove such local actor services. The main idea is that one may assume at the beginning of the message handler the actor invariant and the message precondition. One must then under these assumptions show that the message handler is *valid*. This means that all the field reads and writes are valid (the permissions to these fields are held), all the preconditions are respected for the messages sent and the actor invariant is established at the end. Apart from showing validity, one must show that at least one of the response messages is sent in each possible trace and the where-clause holds. In this case old expressions in the where-clause are evaluated in state at the beginning of the message handler, while all other expressions are evaluated when the response is sent.

A set of rules is defined by the logic with which one can derive (generally non-local) actor services from a set of derived local actor services in a particular program state. One such rule involves the instantiation of universally quantified variables as presented in Section 2.4. One of the most important rules involves the composition of two actor services. We illustrate the rule by example.

Consider the example given in Figure 2. We specify the permission $\textbf{acc}(client.val)$ in the precondition of the manager's getsol message handler making the read of the client.val field valid. We specify the same permission in the worker's sendsol message handler. The remaining permissions to the fields are held in the corresponding actor invariants guaranteeing the validity of the message handlers. We can derive the following local actor service for the manager:

$$\begin{gathered} \underline{M}.\mathsf{getsol}(\underline{C}) \rightsquigarrow \exists W, n.\ W.\mathsf{sendsol}(C, n) \\ \textit{where}\ (\textbf{old}(C.val) = C.val) * n = C.val \end{gathered} \tag{2.2}$$

There are two aspects we have not seen before in this local actor service. First, we use existential quantifiers in the response message for the worker which allows us to abstract over implementation details. Second, we have a non-trivial where-clause. Note that the where-clause is framed (see Section 2.6) since permission to $C.val$ is held in the trigger state (to frame **old**(C.val)) as well as in the response state (to frame C.val). The local actor service clearly holds since C.val is not changed between trigger and response.

This actor service states that whenever the trigger message is received, eventually the response message $W.\mathsf{sendsol}(C, n)$ will be sent for some W and some n. If we can show that in *every future state* (with respect to the current state) for client C and for *all* workers W' and integers n' that whenever the sendsol(C,n) message is received eventually a response message, then we can compose these two properties.

We can derive the following local actor service (which strengthens local actor service (2.1)) for the worker:

$$\underline{W'}.\mathsf{sendsol}(\underline{C}, \underline{n'}) \rightsquigarrow C.\mathsf{sol}(f(n')) \ \textit{where} \ \textbf{old}(C.val) = C.val \tag{2.3}$$

We specify $\textbf{acc}(C.val)$ in the precondition of the client's sol message (essentially sending the permission back to the client) which ensures the where-clause is framed. Since this is a local actor service it holds in every program state: we can show that it holds in every future state for every worker W' and every integer n'. Therefore we may compose (2.2) with (2.3) to get

$$\underline{M}.\mathsf{getsol}(\underline{C}) \rightsquigarrow \exists n.\ C.\mathsf{sol}(f(n)) \ \textit{where} \ (\textbf{old}(C.val) = C.val) * n = C.val$$

Informally, one may derive the where-clause in the composed actor service as follows. Since where-clauses are framed it is guaranteed that all the heap locations mentioned stay the same between the sendsol response message being sent in (2.2) and the sendsol trigger message being received in (2.3). Hence $C.val$ in the where-clause of (2.2) evaluates to the same value as $\textbf{old}(C.val)$ in (2.3). From this observation the remaining reasoning steps are straightforward.

The composition rule uses that if a message is sent to an actor, then it will be eventually received and processed. This holds because of the assumptions introduced in Section 2.1.

Note that the existential quantifier for the worker has disappeared since the worker does not appear any more. This composition rule shows one of the main modularity aspects of the logic, as allows describing end-to-end behaviours, abstracting away what happens in between. Furthermore, these behaviours may themselves be composed again.

The formal composition rule can be found on page 13 of [20].

2.8 Limitations

We present the main limitations of the actor services logic that we address in this thesis. One limitation is that the logic cannot deal well with cases, where one must track the actor's local state over a sequence of multiple messages. There are two options (to the best of our knowledge) to tackle such scenarios in the current logic. One option is to pass information about the actor's state over messages to other actors, even though this information is not used by any of the other actors. This option is unsatisfactory, since it exposes unnecessary information.

Another option is to enforce that the relevant state does not change using immutability predicates. This option is also not optimal, because it restricts the program's behaviour. Furthermore, if one chooses a heap location to be immutable, then it must have the same value from that point onwards forever. Hence there is no way to ever deallocate this heap location, since this would violate the immutability property.

Since this scenario of tracking an actor's state over multiple messages appears often in protocol-based programs, this limitation is relevant for such programs. In Section 3 we illustrate the limitation of the logic by considering a protocol-based program (we show both options that the logic currently provides to tackle the limitation). In Section 4, we show how to lift this limitation for a class of protocol-based actor programs.

Another limitation is that actor services can only have single trigger messages and single response messages (alternative response messages are possible but not the guarantee that more than one response message is sent). We address this limitation in Section 5.

3 Protocol reasoning using the original actor services logic

Consider the protocol-based Akka actor program in Figure 1 which was introduced in Section 1. The corresponding encoding in ActorPL introduced in Section 2.2.2 is shown in Figure 4. The desired response property in the original Akka program is that whenever the manager gets a Query message then eventually the client specified in the Query message will get a Sol message with value $f(n)$ where n is the value specified in the Query message. This response property holds only if one can guarantee the following properties:

1. When a Query message is received by the manager, then the manager will accept this message and will not throw an exception. If the branch were taken where an exception is thrown, then the solution to this query would never be sent.

2. After sending the Compute message, the manager only receives the Result message containing $f(n)$ and not some other Result message. If this were not the case, then potentially the wrong solution would be sent.

The first property states that the manager must only receive expected Query messages. The response property can be expressed by the following actor service in the ActorPL encoding:

$$\underline{M}.\mathsf{query}(\underline{C}, n) \rightsquigarrow C.\mathsf{sol}(f(n)) \tag{M1}$$

In this section we attempt to verify this response property using the original actor services logic. As a result, we will gain an insight into the challenges of verifying such properties modularly in protocol-based actor programs and gain an understanding of the limitations present in the original actor services logic with respect to these programs. In the following approaches we ignore the issue of exposing implementation details in specifications. Instead, we focus on higher-level modularity aspects. In Section 4 we also address the implementation detail issue.

3.1 Ownership transfer

As already discussed in Section 2.2.2, to ensure that the semantics of the encoded ActorPL program reflect the semantics of the original Akka program we need to make sure that only the manager can modify its own state and client fields. We can ensure this by keeping at least partial permission to these fields in the manager's actor invariant at all times.

As a first approach let us put the exclusive permission to both fields in the manager's actor invariant. This way we can be sure that all the field writes in the manager implementation are valid. We can then derive the following local actor service for the manager (assuming the precondition of the compute message sent in the manager implementation is respected):

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow \exists W.\ W.\mathsf{compute}(M, n) \mid \mathsf{fail}()$$

In the response pattern we cannot just provide the compute response message because the query message may be received when the manager is waiting for a result which leads to a failure. This local actor service will not be enough to derive the desired response property because we will never be able to get rid of the fail() branch.

One way we can make sure that the query message will be received at a time when the manager is ready to accept query messages (i.e. its state field evaluates to ACCEPTQUERY) is to provide a precondition that ensures the query message can only be received in such states. This would ensure that there are no unexpected query messages in all program contexts that respect the precondition.

We enable this by using *fractional permissions* [5] which is an extension of the exclusive permissions introduced in Section 2.5. It generalizes accessibility predicates to have the form $\mathbf{acc}(x.f, q)$ where q is between 0 and 1. If q is 1 then this represents the exclusive permission and otherwise it represents partial permission. Exclusive permission is required to write to a location and partial permission is sufficient to read from a location. It is guaranteed that the sum of all fractions for a particular heap location always add up to 1 (we have $\mathbf{acc}(x.f, q_1) * \mathbf{acc}(x.f, q_2) \Leftrightarrow acc(x.f, q_1 + q_2)$ if $q_1 + q_2 \leq 1$, otherwise the state is inconsistent).

To ensure that no one else can modify the field we keep half of the exclusive permission to **this**.state in the manager's actor invariant at all times. The other half permission can be sent around. Hence we define the precondition of the query(c,n) message handler as follows:

$$\mathbf{acc}(\mathbf{this}.\mathsf{state}, \frac{1}{2}) * \mathbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY}$$

Analogously, we define the precondition of the result(res) message handler as follows:

$$\mathbf{acc}(\mathbf{this}.\mathsf{state}, \frac{1}{2}) * \mathbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT}$$

To make sure that the worker can send the result message we define the precondition of the worker's compute(m,n) message handler as:

$$\mathsf{acc}(\mathsf{m.state}, \frac{1}{2}) * \mathsf{m.state} = \mathsf{WAITFORRESULT}$$

Note that with these preconditions all the manager's message handlers are valid (see Section 2.7 for the validity of message handlers), assuming the precondition of the sol message sent to the client is respected. The state field writes are well-defined since the half permission from the precondition together with the half permission from the invariant give the exclusive permission. The precondition of the compute message sent to the worker is respected since half permission to the state field can be given up and it has the correct value. Furthermore, the preconditions guarantee that the manager only receives messages that it expects independent of the context in which the manager is used (as long as one makes sure that all actors in that context respect the preconditions).

So the main idea here is that the manager gives up partial ownership to its own local heap and transfers it to the worker. This partial ownership along with the information about the heap value plays the role of a witness that ensures that the manager is ready to accept a specific message. Without transferring this ownership the worker would have no way of ensuring that the manager will accept a result message. Let us now see what we can derive.

With these preconditions we can prove that the fail() branch is never taken in the query message handler. Hence we can derive the following local actor service for the manager:

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow \exists W.\ W.\mathsf{compute}(M, n)$$

For the worker we can derive the following local actor service:

$$\underline{W}.\mathsf{compute}(\underline{M}, n) \rightsquigarrow M.\mathsf{result}(f(n))$$

Since the existentially quantified variable in the manager's local actor service is a worker and the worker's local actor service is applicable for every worker in future state (see Section 2.7 for the composition of actor services), we can compose the two actor services in any program state to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n))$$

Furthermore, we can derive the following local actor service for the manager since the fail() branch is never taken in the result message handler:

$$\underline{M}.\mathsf{result}(\underline{r}) \rightsquigarrow M.\mathsf{client}.\mathsf{sol}(r)$$

We may instantiate r with $f(n)$ and then compose these two actor services in any program state to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \leadsto M.\mathsf{client.sol}(f(n))$$

While this derived actor service certainly holds, it is not exactly the same as the desired actor service (M1). In our derivation we have lost the connection between $M.\mathsf{client}$ and the initial argument C. Without this connection the response property is not strong. It just states that if the manager receives a query message then eventually some client (we only know that at the time of the response this client is stored in $M.\mathsf{client}$[4]) will get a sol message with value $f(n)$. In the next approach we show one way to recover the connection between $M.\mathsf{client}$ and C.

As a side remark, we did not specify what happens with the half permission to the state field gained by the manager from the reception of the result message. One option would be to transfer it to the client over the sol message (adjusting the precondition of the sol message handler accordingly), which would allow the client to send another query message or to delegate this permission to someone else. Another option would be for the manager to keep this permission. To do the bookkeeping in the manager's actor invariant for this second option one would need more information in the manager which could be gained by adding ghost fields[5].

3.2 A complete but non-modular approach

3.2.1 High-level strategy

In the previous approach we derived

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \leadsto M.\mathsf{result}(f(n))$$

This actor service is not precise enough for our purposes, because we cannot relate the program state when the result response message is received to the program state when the query message was received. To derive the desired actor service (M1) we need to know that $\underline{M}.\mathsf{client}$ evaluated in the response state evaluates to C. The strategy in this approach is as follows. First, we establish the fact that $\underline{M}.\mathsf{client}$ evaluates to C when the manager sends the $\mathsf{compute}$ message to the worker upon receiving the query message. We then make explicit in each of the actor services that $\underline{M}.\mathsf{client}$ does not change.

[4]See Section 2.4 for heap-dependent expressions in actor services.

[5]Ghost fields are fields that are only used for the purpose of verification and which have no effect on the behaviour of the program.

The way this is achieved is by sending partial permission to $\underline{M}.\mathsf{client}$ over the messages, such that each actor that takes part in the response can establish this fact independently.

3.2.2 Derivation

We first define the specifications for the manager and the worker. The manager's specification is given by

Precondition for $\mathsf{query(c,n)}$:

$$\mathbf{acc}(\mathbf{this}.\mathsf{state}, \frac{1}{2}) * \mathbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY}$$

Precondition for $\mathsf{result(res)}$:

$$\mathbf{acc}(\mathbf{this}.\mathsf{state}, \frac{1}{2}) * \mathbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT} * \mathbf{acc}(\mathbf{this}.\mathsf{client}, \frac{1}{2})$$

Manager's actor invariant:

$$\begin{aligned}
&\mathbf{acc}(\mathbf{this}.\mathsf{state}, \frac{1}{2}) * \\
&(\mathbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY} \Rightarrow \mathbf{acc}(\mathbf{this}.\mathsf{client})) * \\
&(\mathbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT} \Rightarrow \mathbf{acc}(\mathbf{this}.\mathsf{client}, \frac{1}{2}))
\end{aligned}$$

The worker's specification is given by

Precondition for $\mathsf{compute(m,n)}$:

$$\mathbf{acc}(\mathsf{m.state}, \frac{1}{2}) * \mathsf{m.state} = \mathsf{WAITFORRESULT} * \mathbf{acc}(\mathsf{m.client}, \frac{1}{2})$$

As discussed in Section 3.2.1 the idea is to send partial permission to the manager's client field over the messages, so that we can prove in each step that the field has not changed. Without this permission, we would not be able to prove it in every step, because someone else may hold the full permission to the field and may hence modify it. This is the reason we specify permission to the client in the precondition of $\mathsf{compute}$, so that the permission can be transferred to the worker. Since the manager needs to give up this permission, it can only retain the remaining (half) permission in the actor invariant, while it is waiting for the result message. This is why the manager does not hold exclusive permission when its state field evaluates to WAITFORRESULT.

The permisson to the client field received by the worker over the compute message has to be transferred back to the manager over the result message, because again we want to ensure that the client field will not change between the worker sending result and the manager receiving result. Hence the permission is specified in the precondition of the result message handler. The precondition for query is the same as in the previous approach.

We can now show the derivation. We can derive the following local actor service for the manager

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow \exists W.\ W.\mathsf{compute}(M, n)\ \ \textit{where}\ M.\mathsf{client} = C$$

Here we establish the fact that M.client evaluates to the client C given in the query message. Note that the where-clause is framed due to the precondition of compute. Next, we derive the following local actor service for the worker

$$\underline{W}.\mathsf{compute}(M, n) \rightsquigarrow M.\mathsf{result}(f(n))$$
$$\textit{where}\ \textbf{old}(M.\mathsf{client}) = M.\mathsf{client}$$

Recall that old expressions are evaluated in the state when the trigger message is received and other expressions are evaluated in the state when the response message is sent (see Section 2.4). Here we establish that the client field does not change between compute being received and the result response message being sent. This is only possible because we send the permission to this field to the worker, otherwise the worker would not be able to establish this fact. We can compose the two derived local actor services to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n))\ \ \textit{where}\ M.\mathsf{client} = C \tag{3.1}$$

Since we established in each step that the client field did not change, we could now establish that when the manager receives the result response message the client field evaluates to the client C when the query was sent. We have now recovered this connection. We want to now compose this actor service with the local actor service that describes the manager's reaction to the result message. Recall that in the previous approach we express this reaction using the local actor service

$$\underline{M}.\mathsf{result}(\underline{r}) \rightsquigarrow M.\mathsf{client}.\mathsf{sol}(r)$$

If we composed this actor service with actor service (3.1), we would not be able to deduce that the client field when the sol message is sent evaluates to

C because we do not know if the field has changed. One option would be to make it explicit that the client field has not changed, but for this we need to transfer permission to the client field over the sol message (otherwise the where-clause is not framed). Instead, we take a cleaner approach. We derive

$$\underline{M}.\mathsf{result}(\underline{r}) \rightsquigarrow \exists C.\ C.\mathsf{sol}(r)\ \textit{where}\ C = \textbf{old}(M.\mathsf{client})$$

Here we express the client to whom the sol message is sent using an existentially quantified variable and state that it evaluates to the same value as the manager's client field when the result message is received. This way it is possible for the manager to retain the permission to the client field and establish the actor invariant. Composing this actor service with actor service (3.1) leads to

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n))$$

This is exactly the actor service we wanted to derive in the first place.

3.2.3 Discussion of approach

While we have achieved our goal of verifying the main response property for the actor program, there is an issue with the outlined approach related to modularity. In the precondition of the worker's compute message handler partial permission to the manager's state and client fields are needed. One can motivate the presence of the partial permission to the manager's state field from the worker's viewpoint because the worker wants to at some point send a result message to the manager and hence the worker needs a guarantee that this message will not lead to a failure (i.e. the message is accepted). So this permission has a clear meaning for the worker[6]. However, from the worker's viewpoint it is not clear why it requires permission to the manager's client field. The only motivation for the worker to get this is because the precondition of the result message handler requires that permission.

To understand why this is even more of an issue, consider the following scenario. Suppose the worker W does not immediately send the result message to the manager upon receiving the compute message. Instead, W sends a message to some subworker S which then sends the computed result back to W at which point W sends the result to the manager. A message sequence diagram showing this scenario is given in Figure 6 on page 51.

To derive the same response property using the outlined approach the worker would have to send the manager's client field permission to S which sends it back to W. This is an issue because S should not have to know

[6]We ignore the fact that the implementation details are being exposed for now and come back to it in Section 4.

anything about the manager, since S never interacts with the manager in any form. If S needed to know about the manager then this would not be modular, because a change in the manager would could affect S even though they are independent. It should be possible to use S in scenarios which do not involve the manager at all. Note that since W knows the value of the state field, the corresponding permission would not have to be sent around to derive the desired actor service. As protocols involve more actors one would have to send permissions to multiple fields that do not change across the whole topology which makes this an even bigger issue.

3.3 A modular but incomplete approach

3.3.1 High-level strategy

Consider actor service

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n))$$

which was derived in Section 3.1. As already stated before in Section 3.2.1 the main issue is that we cannot relate M.client in the response with C. In the approach outlined in Section 3.2 we showed how to recover this connection between M.client and C by transferring permission to M.client. However, as we discussed in the same section, the approach breaks modularity.

We follow a different strategy in this approach. First, we establish that M.client evaluates to C when the manager sends the compute message. Next, we make sure that at the point when M.client is set to C (before the compute message is sent) that it is not going to possible to ever change M.client. This means we ensure that M.client becomes immutable. Hence in every future state after the compute message is sent, we can be sure that M.client evaluates to C. We discuss the consequences of this decision at the end of the section (in particular, we are restricting the program).

3.3.2 Derivation

We use the same preconditions as in the approach presented in Section 3.1, where partial permission to the manager's client field is not given away. The main difference lies in the manager's actor invariant. We define it as follows:

$$\begin{aligned}
&\left(\begin{array}{l}\textbf{acc}(\textbf{this}.\mathsf{client}) * \textbf{this}.\mathsf{client} = \mathsf{null} * \textbf{acc}(\textbf{this}.\mathsf{state}, \tfrac{1}{2}) * \\ \textbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY}\end{array}\right) \vee \\
&\quad\left(\begin{array}{l}\textbf{immut}(\textbf{this}.\mathsf{client}) * \textbf{acc}(\textbf{this}.\mathsf{state}, \tfrac{1}{2}) * \\ \textbf{this}.\mathsf{client} \neq \mathsf{null} * \textbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT}\end{array}\right) \vee \\
&\qquad\qquad\left(\begin{array}{l}\textbf{immut}(\textbf{this}.\mathsf{client}) * \textbf{acc}(\textbf{this}.\mathsf{state}) * \\ \textbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY}\end{array}\right)
\end{aligned}$$

The actor invariant consists of a disjunction of three mutually exclusive subassertions (only one of the three subassertions can hold at any given point). The first subassertion describes the manager when it is spawned. In this state client field may still be mutated since no query has been received. The second subassertion describes the manager when the first query message has been received and the manager is waiting for the result message. In this state the client field is immutable (following the strategy described in Section 3.3.1). The third subassertion describes the manager when the result message has been sent. In this state the exclusive permission to the state field is, ensuring that no one can send any query messages anymore, since we cannot mutate the client field at this point anymore.

Next, we discuss the validity of the message handlers, which is not completely trivial due to the complexity of the actor invariant. Suppose the manager receives a query message. The precondition of query provides half permission to the state field where it evaluates to ACCEPTQUERY. Hence right at the beginning of the query message handler one can deduce that only the first subassertion of the actor invariant can hold (the other two are inconsistent together with the precondition). Hence at the beginning of the query message handler we have

$$\begin{array}{l}\textbf{acc}(\textbf{this}.\mathsf{client}) * \textbf{this}.\mathsf{client} = \mathsf{null} * \textbf{acc}(\textbf{this}.\mathsf{state}) * \\ \textbf{this}.\mathsf{state}{=}\mathsf{ACCEPTQUERY}\end{array}$$

The write to the client field in this message handler is valid. The precondition of the sent compute message is respected, since just half permission to the state field must be given up which is obtained in the precondition and the state field must evaluate to WAITFORRESULT, which it does. At the end of the message handler one can transform the exclusive permission to the client field into an immutability permission and re-establish the actor invariant.

Next, suppose the manager receives a result message. Using similar reasoning as in the query case one can show that at the beginning of the query message handler the following assertion holds

$$\begin{aligned}&\textbf{immut}(\textbf{this}.\mathsf{client}) * \textbf{acc}(\textbf{this}.\mathsf{state}) * \\ &\textbf{this}.\mathsf{client} \neq \mathsf{null} * \textbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT}\end{aligned}$$

After changing the state field the actor invariant can be established. As mentioned before, from this point onwards it is not possible for anyone to send a query message since the exclusive permission to the state field is always held in the actor invariant.

Now that we have shown that the manager's message handlers are valid with respect to the chosen actor invariant and preconditions, we derive the desired response property. We have the following local actor service for the manager

$$\begin{gathered}\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow \exists W.\ W.\mathsf{compute}(M, n) \\ \textit{where } \textbf{immut}(M.\mathsf{client}) * M.\mathsf{client} = C\end{gathered}$$

Here we establish that the client field evaluates to C given in the query trigger message. Furthermore, we establish that at the point the compute message is sent the client field is guaranteed to be immutable. The where-clause is framed since it contains the immutability predicate for the client field. We have the following local actor service for the worker

$$\underline{W}.\mathsf{compute}(\underline{M}, n) \rightsquigarrow M.\mathsf{result}(f(n))$$

We can compose the these two local actor services to get

$$\begin{gathered}\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n)) \\ \textit{where } \textbf{immut}(M.\mathsf{client}) * M.\mathsf{client} = C\end{gathered}$$

This is possible because we know from the where-clause of the first local actor service that when the compute message is sent M.client is immutable and evaluates to C, hence this also holds in every future state. The response message is sent in a future state, hence we can carry along the fact. So the immutability feature allows us to retain the fact without sending any permission for the client field to the worker. We have the following local actor service for the manager

$$\underline{M}.\mathsf{result}(\underline{r}) \rightsquigarrow M.\mathsf{client.sol}(r)$$

We can compose the previous actor service with an instantiation of this local actor service to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n))$$

which is the desired response property. In this step we again use the immutability of M.client.

3.3.3 Discussion of approach

The outlined approach is modular in the following sense. Consider the scenario discussed in Section 3.2.3 for which the message sequence diagram is given in Figure 6 on page 51. In this scenario the worker sends a message to the subworker and uses the result received from the subworker for the result sent to the manager. In the outlined approach the subworker would never get to know about the manager. Permission to the manager's fields does not have to be sent to the subworker. Hence the subworker can be used in contexts where the manager may not even exist. We conclude that immutability can be used to achieve modularity.

However, this approach has severe drawbacks. The manager can only process a single `query` message since it cannot mutate its `client` field after the first `query`. Hence the manager cannot be reused. This means we have proved a property about a restricted program that does not reflect the semantics of the original program which was for the manager to deal with multiple `query` messages. Another disadvantage is that there is no way for the manager to deallocate the `client` field, once the first `query` message has been received as explained in Section 2.8.

3.4 Conclusion

The presented approaches show the trade-off that needs to be made in the original actor services logic between modularity and a more general result (reuse of actors, support for deallocation) in actor-based protocol programs (as hinted at in Section 2.8). There does not seem to be an easy way to get the best of both worlds without extensions to the logic. In the next sections we present an approach that gets the advantages of both presented approaches for an interesting class of protocol-based actor programs.

4 Modular session-based reasoning in actor programs

In this section we present our main technique for the verification of response properties in protocol-based actor programs. First, we introduce the notion of a *session*, which describes a specific interaction in the actor governed by a protocol, and motivate its use in our setting. We then develop initial machinery to support sessions in an extended version of the actor services program logic. Next, we present fundamental challenges and corresponding solutions in this setting related to tracking information during a session and providing modular specifications. We then outline how the developed technique can be potentially generalized to an even larger class of programs. Finally, we identify a class of programs for which our technique is effective.

4.1 Motivating sessions

In Section 3 we tried to use the original actor services logic to verify the actor service (M1) for the actor program given in Figure 4. The response property states that whenever the manager gets a query message, then eventually the client specified in this message will get a correct result message. The response property holds if one can guarantee the absence of unexpected messages received by the manager and that only the correct result message is received. We managed to capture these properties by adjusting the preconditions of the message handlers, restricting when a message can be sent. However, we could only derive the actor service if we either sacrificed modularity or we further restricted the program using immutability.

The heart of the problem in all the presented approaches lies in the composition of the following two actor services

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow \exists W.\ W.\mathsf{compute}(M, n)$$

$$\underline{W}.\mathsf{compute}(\underline{M}, n) \rightsquigarrow M.\mathsf{result}(f(n))$$

which can be composed to

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n)) \tag{4.1}$$

Actor service (4.1) states that after the manager receives a query message, it will eventually receive the correct result message. This is an important part of the behaviour but does not precisely describe the situation. The problem is that actor services do not tell us when the response will be sent/received. In this particular case by just looking at the actor service we do not know which result message is meant: maybe it is a result message corresponding to a different query that was received later by the manager. As a result, we do not know what M.client (which is set to the client C once the trigger message is received) evaluates to when the result message is received. We would like to have an actor service that expresses the following:

> *If the manager receives a* query *trigger message, then eventually the manager will receive the* result *message with value $f(n)$. Furthermore, we have that the* result *message corresponds to the result that the manager is expecting with respect to the* query *trigger message (hence no further* query *or* result *messages are accepted by the manager before it receives this* result *message).*

If we had such an actor service, then we would know which result message is meant. However, the actor service would not be sufficient to ensure that M.client has not changed, because there could be message handlers other than result and query in the manager which could change M.client. We could get this guarantee, if we additionally ensured by just considering the manager's implementation, that until the expected result message is received, the manager's state required for the query is not modified. Such reasoning would be fundamentally different to the approach in Section 3.2 where we explicitly talk about M.client in each step which breaks modularity. Here we need not mention M.client outside of the manager. To enable expressing such an actor service, we need a way to express that a result message corresponds to a query message. One way to do this is to use the notion of a session.

A session in our setting represents a concrete interaction governed by an abstract protocol from the point of view of a single actor. It starts at some point and potentially finishes at another point. The protocol states how the session progresses. For a specific protocol there can be multiple sessions for a single actor. In our setting we only permit a single session to be active at any given point for a particular protocol in an actor. Once a session governed by a protocol ends, another session for the same protocol can be started in the actor. However, multiple sessions corresponding to different protocols may be active simultaneously in an actor.

In our example, the interaction starting from the moment the manager can receive a query message until it sends the corresponding sol message to the client is one session that is governed by a specific protocol which we call *SM*. Once the manager has sent the sol message, the current session is finished after which another session governed by *SM* can be started (however, one can choose to start the session at any point later as well in general). In the session first a query message is expected and once this message is received a result message is expected next. We represent these stages using *protocol states*. Let us call the protocol states **Q** (where queries are received) and **R** (where results are received). The *SM* protocol dictates that the session progresses from **Q** to **R**. We call the current protocol state of a session the *session state*. To distinguish different sessions we use an abstract *session identifier*. Using this idea of a session we can rephrase what our actor service should express:

> *If the manager receives a* query *trigger message, then eventually the* result *message with value $f(n)$ will be received. Furthermore, the following holds:*
>
> - *When the trigger message is received, the manager's session governed by SM is in the **Q** state.*
> - *When the response message is sent, the manager's session governed by SM is in the **R** state.*
> - *The session identifiers of the sessions governed by SM in the trigger and response states are identical (i.e. they refer to the same session).*

This description is clearer and more precise than the previous one. Suppose we can extend the actor services logic such that one can derive such an actor service. How can we deduce that M.client has not changed between the trigger and response states? This should become possible if we enrich the description of a protocol using a *protocol invariant* that describes for each session state what properties hold and that may relate different session states. In this particular case the protocol invariant could state that the M.client field does not change during a session once it has reached the **R** state (but once the session finishes it may be modified again). If one knows this, then answering the question becomes simple.

Note that in this example the session state directly corresponds to the value of the state field. If the state field evaluates to ACCEPTQUERY, then the session state is **Q** and otherwise if the state field evaluates to WAITFORRESULT, then the session state is **R**. So the session state and therefore the session cannot be modified by the manager without the exclusive permission to the state field. This means the partial permission to the state field that is sent around in the approaches presented in Section 3 corresponds to *partial session ownership*. Without this ownership, the manager cannot progress or finish its session. To make this more direct, instead of passing around permission to the state field[7] which exposes implementation

[7]Recall that we needed to pass this permission around to be sure that messages sent

details, we can pass partial session ownership that is represented by a *session predicate* abstracting over implementation details.

4.2 Introducing sessions in the actor services logic

In Section 4.1 we motivated the concept of a session and introduced basic concepts informally. Next, we make these concepts explicit and show how they can be incorporated into the actor services program logic. This incorporation, for now, is done by mainly altering how message handlers are verified.

4.2.1 Protocol descriptions

Sessions are specific to a *single* actor and they are parameterized by *protocol descriptions* for that actor type. Protocol descriptions describe part of the behaviour of an actor during a session that it governs. A protocol description ρ is defined by

- the actor type τ_ρ for which the protocol description is defined
- a set of *protocol states* PState_ρ
- a *transition relation* $\sqsubset_\rho$ that is a strict partial order on the protocol states that specifies to which states an actor may move during a session
- a *protocol invariant* $\mathit{Inv}_\rho(s)$ that is a function from protocol states to self-framing assertions[8], where $\mathit{Inv}_\rho(s)$ holds in the actor if the session is in state s

In the example of Figure 3 the sessions described for the manager are governed by protocol description SM (hence $\tau_\rho = \mathsf{Manager}$). We have

$$\mathit{PState}_{\mathit{SM}} = \{\mathbf{Q}, \mathbf{R}\}$$

where $\mathbf{Q}$ is the state where the manager accepts query messages and $\mathbf{R}$ is the state where the manager accepts result messages. The transition relation is given by

$$\mathbf{Q} \sqsubset_{\mathit{SM}} \mathbf{R}$$

to the manager are going to be accepted.

[8]See Section 2.5 for a definition of self-framing assertions.

indicating that the manager can only move from $\mathbf{Q}$ to $\mathbf{R}$. A potential protocol invariant is (**this** refers to the actor controlling the session)

$$\begin{aligned} Inv_{SM}(s) :=\, & \textbf{acc}(\textbf{this}.\mathsf{state}) * \textbf{acc}(\textbf{this}.\mathsf{client}) * \\ & (s = \mathbf{Q} \Rightarrow \textbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY}) * \\ & (s = \mathbf{R} \Rightarrow \textbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT}) \end{aligned}$$

As we will see, this invariant is not strong enough to verify the desired actor service. Later we will present a protocol invariant that allows verifying the desired actor service.

Note that there is no need to represent initial and final states. As we will see, one may start in any state (as long as one can guarantee the protocol invariant) and finish in any state. Furthermore, one may progress to any state that is later in the transition relation. This gives greater flexibility.

4.2.2 Session ownership and session predicates

In the example given in Figure 4 when, for example, the worker sends a result message to the manager, then we need to make sure that this message will be accepted by the manager so that it does not lead to a failure. Another way of phrasing this is that the worker needs to be sure when sending the result message that the session state of the session in the manager governed by *SM* is given by $\mathbf{R}$ and the session will not progress until the result message is received.

We enable such reasoning by introducing the notion of *session ownership*. To allow other actors to send messages that are part of the session (such as the worker sending the result message) we introduce *session predicates* that represent partial ownership of the session (the actor controlling the session always must own part of the session, hence full session ownership is never given up). For a session governed by protocol ρ there is a unique session predicate given by $\rho(a)$ where a is the actor that controls the session (its actor type must be the same as τ_ρ).

The session predicate is generated right when a session is started (see Section 4.2.5 for details on session initialization). An actor may give up the session predicate to another actor. This represents the transfer of partial session ownership to that actor. We ensure that without the session predicate the corresponding session cannot progress or finish (see Section 4.2.4 for details on session progress and Section 4.2.5 for details on finishing a session).

Another part of the session permission is held with the actor controlling the session (i.e. the manager in the example). It is contained in the unique *session token* $\rho^{tok}(a)$ and it is also required to progress and finish a session.

This token cannot be sent around and cannot be part of any specification. The reason we model this token explicitly is technical; for more details see Section 4.2.3. One can think of the session token as being part of the protocol invariant, even though we do not explicitly mention it in the protocol invariant. The session token and the session predicate are both handled like exclusive permissions with respect to the separating conjunction, which were introduced in Section 2.5.

As an example, the manager can send its session predicate for the session governed by the SM protocol to the worker over the compute message. This can be done by adding $SM(\mathsf{m})$ to the precondition of the compute message handler. Hence if the worker holds the session predicate $SM(\mathsf{m})$, then the manager m cannot progress or finish this session.

Finally, the remaining part of the permission is held in the *session finalization permission*. This permission is required to finish a session, but not to progress the session. We will use this permission in Section 4.8 to give other actors a guarantee that a session will not finish until a specific interaction sequence has completed. We use *counting permissions* [4] to be able to distribute this finalization permission. This means there is a unique source permission $\rho^F(a, k^-)$ that is always held in the actor a controlling the session, where k is the number of *access permissions* that have been given away. The permission holding k' access permissions is given by $\rho^F(a, k'^+)$. We have

$$
\begin{aligned}
&\rho^F(a, k_1^+) * \rho^F(a, k_2^+) \Leftrightarrow \rho^F(a, (k_1 + k_2)^+)\\
&\rho^F(a, k_1^-) * \rho^F(a, k_2^-) \Leftrightarrow \mathit{false}\\
&\rho^F(a, k_1^-) * \rho^F(a, k_2^+) \Leftrightarrow \begin{cases} \rho^F(a, (k_1 - k_2)^-) & \mathit{if}\ k_1 \geq k_2 \\ \mathit{false} & \mathit{otherwise} \end{cases}
\end{aligned}
$$

When the session is started $\rho^F(a, 0^-)$ is held, indicating that no access permissions have been given away, hence this denotes the full finalization permission. The source permission must always be contained in the protocol invariant. In examples, if we do not give away access permissions, then we will not explicitly mention these finalization permission in the protocol invariant. In such cases, if we give the protocol invariant definition $Inv_\rho(s) := A$, then we actually mean $Inv_\rho(s) := A * \rho^F(\textbf{this}, 0^-)$.

4.2.3 Session attributes and attribute functions

Sessions have two attributes that describe it: a *session identifier* and a *session state*. The session identifier is a unique identifier (with respect to all session identifiers of sessions governed by the same protocol in the same actor) that distinguishes the session from other sessions governed by the same protocol in the same actor. This session attribute stays the same throughout the session. The session state specifies the protocol state that the session is in currently, and it changes as the session progresses. To be able to talk about these attributes in assertions we introduce *session attribute functions* parameterized by the protocol description ρ governing the session and actor a controlling session. $\iota_\rho(a)$ is a session attribute function which evaluates to the session identifier of the current session governed by protocol description ρ in actor a. $state_\rho(a)$ is a session attribute function which evaluates to the session state of the current session governed by protocol description ρ in actor a. Since for each protocol description and actor only one session can be active, it is clear which session is meant in these definitions (assuming one such session is active).

While we have now defined functions that return the attribute values, we must still discuss when one can use these functions in assertions. For example, if we have an assertion $state_{SM}(m) = \mathbf{Q}$ in a particular state then we must be sure that the session is active and its session state for m really is $\mathbf{Q}$ and it is not possible that m progresses the session, while we are in a state where the assertion holds. If we cannot guarantee this, then the assertion is meaningless.

The idea is to use the notion of session ownership introduced in Section 4.2.2. If one owns the corresponding session predicate or session token, then one can be sure that the session is active and cannot progress or finish. Hence, in such cases the session identifier and the session state of the current session cannot change, therefore it is fine to use the attribute functions in assertions.

If one owns a corresponding session finalization access permission, then one can be sure that the session is active and cannot finish. Hence in such cases the session identifier of the current session cannot change, therefore it is fine to use the identifier attribute function in assertions.

Formally, we say that $\iota_\rho(a)$ is *framed* by the corresponding session predicate $\rho(a)$ as well as by the corresponding session token $\rho^{tok}(a)$ and by a non-zero session finalization access permission amount $\rho^F(a, k^+)$ $(k > 0)$ (any of the three permissions are enough to frame it). $state_\rho(a)$ is framed by the corresponding session predicate $\rho(a)$ as well as the corresponding session token $\rho^{tok}(a)$ (one of the two is enough to frame it).

Using this framing definition for these functions, we can easily generalize the framing definition for expressions given in Section 2.5 to expressions containing such functions. Also note that since we can think of the session token as being part of the protocol invariant, we may consider expressions

$\iota_\rho(this)$ and $state_\rho(this)$ as framed in the corresponding protocol invariant.

Here we can see the motivation for explicitly representing the session token. If an actor controlling the session sends a session predicate away, it still might want to express properties using these functions. Hence one requires something that frames the functions, which in our case is the session token.

4.2.4 Associating messages to protocols and session progress

Message handlers in an actor can now be explicitly associated with protocols. The protocol definition itself does not say which messages are part of the protocol, it just specifies constraints in terms of abstract protocol states. In our setting we permit a message handler to be part of at most one protocol[9]. If a message handler is part of protocol ρ, then we require that its precondition holds the session predicate $\rho(this)$. For example, we can provide the following valid precondition for the manager's result message handler

$$SM(this) * state_\rho(this) = \mathbf{R}$$

The reasoning behind this is that if a message handler is associated with protocol ρ, then it must be ensured in our setting that this message is sent while the corresponding session is active and the session must not be progressed until the message is received. The only way an actor sending the message can guarantee this is by sending partial ownership of the session using the session predicate. Note that the presented precondition is self-framing since $SM(this)$ frames $state_\rho(this)$ as explained in Section 4.2.3. In general a message can be sent in multiple session states during the same session (however, one must adjust the precondition to allow these different session states).

An actor controlling a session can only progress a session governed by protocol ρ in message handlers that are associated with ρ. Furthermore, if a message handler is associated with ρ then the actor must progress or finish the session inside that message handler. Suppose message handler m is associated with protocol ρ. At the beginning of the message handler one may assume

- the precondition
- the actor invariant

[9]If one allowed a message handler to be part of multiple protocols, then one would have to be able to identify in the body which protocol the received message is part of (assuming that each concrete message can only be part of a single protocol). We believe that it is possible to support this using more machinery, but since it is orthogonal to the core of our work, we do not support it.

- the session token
- the protocol invariant $\mathit{Inv}_\rho(\mathit{state}_\rho(\mathit{this}))$ (note that $\mathit{state}_\rho(\mathit{this})$ is framed by the session token)

Then at some point in the implementation the session must be progressed or finished. To progress a session one must provide

- the session predicate and the session token
- a protocol state s^* for which $\mathit{state}_\rho(\mathit{this}) \sqsubset_\rho s^*$ holds

After the session has progressed the session state is given by state s^* and the session identifier is the same as before. At the end of the message handler one must guarantee the actor invariant in any case and if one has chosen to progress the session (instead of finishing it), one must guarantee

- the session token to make sure that session ownership is retained by the actor controlling the session
- $\mathit{Inv}_\rho(s^*)$ to make sure the protocol invariant is re-established

Using these conditions we can extend the definition of validity (introduced in Section 2.7) of message handlers associated with protocols. Note that only message handlers associated with protocol ρ ever get hold of the protocol invariant for ρ.

One important point is that the session predicate associated with the session that is received over the precondition cannot be sent to other actors before the session is progressed or finished inside the message handler. This avoids other entities getting access to session predicates in session states that have already been used to send a message associated with the same protocol in the same actor. We can support this by putting the session predicate under a modality. Under this modality the session behaves the same as usual, but one cannot transfer it over messages. If the session has been progressed, one gets hold of the session predicate.

4.2.5 Starting and finishing sessions

For each protocol description ρ defined in an actor a, we introduce a unique *spawn token* $\rho^{\mathit{SPAWN}}(a)$. At the beginning of the corresponding constructor all these spawn tokens are made available. To spawn a session for ρ in actor a one must give up $\rho^{\mathit{SPAWN}}(a)$ and choose an initial protocol state to move to (this can be done inside any message handler or in the constructor and one may choose an arbitrary protocol state). In return one gets hold of the corresponding session predicate, the session token and the full session finalization permission, as well as the knowledge that the session state is in the state that one picked. At the end of the message handler (or constructor) one must then ensure the protocol invariant for the state that was chosen.

One may finish session in any message handler. For this one must give up the corresponding session predicate, the session token and the full session finalization permission. In return one again gets hold of the corresponding spawn token. Note that this ensures that there can be at most one session running for a protocol description ρ in an actor a at any given time.

4.3 Basic reasoning in actor services using sessions

In Section 4.2 we introduced sessions into the actor services program logic. Next, we illustrate how we can derive actor services involving sessions using the example given in Figure 4. First, we need to provide the specifications of the actors (i.e. the preconditions, actor invariants, protocol definitions). For the manager we assume the protocol definition *SM* as already introduced in Section 4.2.1. As we will see, the defined protocol invariant is not strong enough to verify the desired actor service but it is enough to showcase some of the main points when deriving actor services involving sessions. We define the precondition of the manager's query message handler as

$$\mathit{SM}(\textsf{\textbf{this}}) * \mathit{state}_{\mathit{SM}}(\textsf{\textbf{this}}) = \mathbf{Q}$$

This requirement essentially states that one needs to show that the manager's session governed by *SM* is in state **Q**. Analogously, we define the precondition of the manager's result message handler as

$$\mathit{SM}(\textsf{\textbf{this}}) * \mathit{state}_{\mathit{SM}}(\textsf{\textbf{this}}) = \mathbf{R}$$

Since the manager wants to send partial ownership of the session governed by *SM* to the worker, we define the precondition of the worker's compute message handler as

$$\mathit{SM}(m) * \mathit{state}_{\mathit{SM}}(m) = \mathbf{R}$$

The manager's message handlers are all valid since they may assume the protocol invariant in the corresponding session states and the protocol invariant contains the required permissions in all states.

Note that we do not expose any implementation details in these preconditions as opposed to the approaches in Section 3. Additionally, we need not deal with explicit fractional permissions here. We can derive the following local actor service for the manager

$$\underline{M}.\textsf{query}(\underline{C}, \underline{n}) \rightsquigarrow \exists W.\ W.\textsf{compute}(M, n)\ \textit{where}\ \textsf{\textbf{old}}(\iota_{\mathit{SM}}(M)) = \iota_{\mathit{SM}}(M)$$

Note that the where-clause is framed, since the session predicate $SM(M)$ is held in the precondition of the query trigger message and in the compute response message. Also we can show that the fail() branch is never taken due to the protocol invariant making the connection between the state field and the session state.

This actor service states that when a query message is received by the manager then eventually a compute message is sent and we have that the session in the manager governed by SM is the same in the trigger as well as in the response state. So we are now reasoning about sessions in actor services without changing the meaning of an an actor service. We just added building blocks which can be now used to talk about sessions in the where-clauses of actor services. Furthermore, we can derive the following local actor service for the worker

$$\underline{W}.\mathsf{compute}(\underline{M}, n) \rightsquigarrow M.\mathsf{result}(f(n)) \text{ } \textit{where } \textbf{old}(\iota_{SM}(M)) = \iota_{SM}(M)$$

This actor service also states the session is still the same (the where-clause does not say anything about the session state, but we know the session state from the preconditions).

We may now compose these two local actor services to derive

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n)) \text{ } \textit{where } \textbf{old}(\iota_{SM}(M)) = \iota_{SM}(M)$$

This actor service now makes explicit that the session at the time the query trigger message is received by the manager is the same as when the result response message is sent to the manager. Hence the result response message corresponds to the query trigger message. Actually, this actor service describes the actor service that we wanted to express in Section 4.1.

There is one remaining issue: How do we learn that M.client in the response state evaluates to the client C in the trigger message? Without answering this question we will not be able to derive the desired response property. The current protocol invariant does not help us do this. We adjust the protocol invariant to state the following: if the session state is **R** then we have that M.client evaluates to the client specified in the query message received in session state **Q** (of the same session). In the next section, we introduce expressions which allow us to state such properties.

4.4 Session events and environment expressions

Consider a session governed by protocol description ρ in an actor a. As we discussed in Section 4.2.3, such a session has a session identifier i, which no other session has that is governed by the same protocol in the same actor. Furthermore, if the session progresses to a new session state s then it is guaranteed due to the strict ordering on the session states that the session was never in this state before. Additionally, the session can only be progressed upon reception of a message m associated with ρ in a.

A session event E is a tuple (ρ, a, i, s, m) and it defines the event right when message m is received during the session identified by i in the session state s, where the session is governed by protocol ρ in a. We require that m is associated with ρ. From the above observations we can conclude that every session event occurs at most once during a session.

Session events correspond to messages that are sent to the actor controlling the session at a particular point. Such a message essentially corresponds to the *environment of the session* passing data to the session (over the message arguments). In the example given in Figure 4 the manager has a session event that corresponds to the query message being sent in the **Q** state of the session governed by *SM*. In this message a client reference is sent along and in this case the manager stores this client in its local heap. To be able to state properties about data received in these session events we introduce *environment expressions*.

An environment expression is of the form $\varphi_\rho(a, i, s, m(y; \overrightarrow{x_i}), e)$. It denotes the expression e evaluated in the state corresponding to session event (ρ, a, i, s, m). $\overrightarrow{x_i}$ and y are bound variables, where $\overrightarrow{x_i}$ represent the formal arguments of the message m and y represents the receiving actor of m (any names can be chosen for these bound variables). e must be dependent only on $\overrightarrow{x_i}$ and y (hence $\overrightarrow{x_i}$ and y may occur as free variables in e). For example, consider the environment expression

$$\varphi_{SM}(\mathsf{mgr}, \iota_{SM}(\mathsf{mgr}), \mathbf{Q}, \mathsf{query(y;client,n)}, \mathsf{client})$$

This environment expression evaluates to the client that the manager mgr received over message query in session state **Q** of the current session governed by protocol description *SM*. Note how one may refer to the argument received in that session event by using the bound variables (in this case the bound variables are the receiving actor y and the arguments query,n).

We often omit the bound variable representing the receiving actor when we do not require it in our environment expression. Hence we could also write the previous environment expression as

$$\varphi_{SM}(\mathsf{mgr}, \iota_{SM}(\mathsf{mgr}), \mathbf{Q}, \mathsf{query(client,n)}, \mathsf{client})$$

One cannot denote arbitrary expressions in environment expressions; it must be ensured that the environment expression is framed. The environment expression $\varphi_\rho(a, i, s, m(y; \overrightarrow{x_i}), e)$ is framed if a, i, s are framed and if the precondition of m instantiated with arguments $\overrightarrow{x_i}$ and receiving actor y frames e. The intuition why we require this framing definition here is that we can only talk about the part of the data handed over by the environment for which the permissions exist in the precondition.

To enable reasoning about environment expressions when proving local actor services we permit the following. At the beginning of a message handler implementation m associated with protocol ρ with formal arguments $\overrightarrow{z_i}$ one may assume that

$$\varphi_\rho(\textbf{this}, \iota_\rho(\textbf{this}), state_\rho(\textbf{this}), m(y; \overrightarrow{x_i}), e) = e[\overrightarrow{z_i}/\overrightarrow{x_i}][\textbf{this}/y]$$

if the environment expression and $e[\overrightarrow{z_i}/\overrightarrow{x_i}][this/y]$ are framed by the permissions held at the beginning of the message handler. $a[b/c]$ denotes the substitution of all occurrences of c in a by b.

The idea here is that when message handler m (which is associated with ρ) is invoked, we know that the beginning of the message handler body represents the session event given by $(\rho, \textbf{this}, \iota_\rho(\textbf{this}), state_\rho(\textbf{this}), m)$. Hence we know that the expression denoted by

$$\varphi_\rho(\textbf{this}, \iota_\rho(\textbf{this}), state_\rho(\textbf{this}), m(y; \overrightarrow{x_i}), e)$$

is evaluated in the state at the beginning of the message handler body. If the formal arguments of the message handler m are given by $\overrightarrow{z_i}$, then we know that these are the concrete arguments that are received over this session event and **this** is the receiving actor. Therefore we can instantiate these values in e to get an expression that evaluates to the same value.

4.5 Proving the client-manager-worker example

We now use the environment expressions introduced in Section 4.4 to prove the desired response property in the example given in Figure 4. We use protocol description *SM* for the manager as defined in Section 4.2.1 but we define the protocol invariant as

$$\begin{aligned} Inv_{SM}(s) := {} & \textbf{acc}(\textbf{this}.\mathsf{state}) * \textbf{acc}(\textbf{this}.\mathsf{client}) * \\ & (s = \mathbf{Q} \Rightarrow \textbf{this}.\mathsf{state} = \mathsf{ACCEPTQUERY}) * \\ & \left(\begin{array}{l} s = \mathbf{R} \Rightarrow \textbf{this}.\mathsf{state} = \mathsf{WAITFORRESULT} * \\ \textbf{this}.\mathsf{client} = \varphi_{SM}(\textbf{this}, \iota_{SM}(\textbf{this}), \mathbf{Q}, \mathsf{query(c,n)}, \mathsf{c}) \end{array} \right) \end{aligned}$$

This states that if the session is in state $\mathbf{R}$ (i.e. after the manager has received the query message and sent the compute message) then **this**.client evaluates to the client provided by the query message in the $\mathbf{Q}$ state of the same session.

Note that $\iota_{SM}(this)$ is framed in the protocol invariant since the controlling actor always has ownership of the session (it always holds the session token, which we do not explicitly mention in the protocol invariant). Hence the provided protocol invariant is self-framing. One can easily establish the protocol invariant for state $\mathbf{R}$ when progressing the session in the query message handler using the reasoning provided in Section 4.4. As already shown in Section 4.3 we can derive the following actor service

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n)) \ \ \textit{where}\ \textbf{old}(\iota_{SM}(M)) = \iota_{SM}(M) \tag{4.2}$$

We know that the query trigger message corresponds to a session event (as introduced in Section 4.4). Furthermore, we know by just inspecting the actor service that the client that is passed over this session event evaluates to C. Hence we can rewrite the actor service to

$$\begin{array}{l} \underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n)) \\ \textit{where} \begin{array}{l} \textbf{old}(\iota_{SM}(M)) = \iota_{SM}(M) \, * \\ \varphi_{SM}(M, \textbf{old}(\iota_{SM}(M)), \textbf{old}(state_{SM}(M)), query(c, n), c) = C \end{array} \end{array} \tag{4.3}$$

This environment expression is framed since the required session predicate to frame $\textbf{old}(\iota_{SM}(M))$ and $\textbf{old}(state_{SM}(M))$ is held in the precondition of the query message handler.

In general, we add a further rewrite rule to the actor services logic. The idea is that whenever one has an actor service of the form

$$e.m(\overrightarrow{e_i}) \rightsquigarrow e'.m'(\overrightarrow{e'_i}) \ \textit{where}\ A \tag{4.4}$$

where m is associated with protocol ρ then one may rewrite the actor service to

$$\begin{array}{l} e.m(\overrightarrow{e_i}) \rightsquigarrow e'.m'(\overrightarrow{e'_i}) \\ \quad \textit{where}\ A * \left(\begin{array}{l} \varphi_\rho(\textbf{old}(e), \textbf{old}(\iota_\rho(e)), \textbf{old}(state_\rho(e)), m(y; \overrightarrow{x_i}), e^*) = \\ \textbf{old}(e^*[\overrightarrow{e_i}/\overrightarrow{x_i}][e/y]) \end{array} \right) \end{array}$$

if the environment expression and $\textbf{old}(e^*[\overrightarrow{e_i}/\overrightarrow{x_i}][e/y])$ are framed. e^* is an expression in terms of the formal arguments $\overrightarrow{x_i}$, and since in the actor service we know which parameters are provided for the session event we can provide an expression in terms of these parameters that evaluates to the same value as the corresponding environment expression (assuming the framing conditions hold). Note that in our example $\textbf{old}(M)$ and $\textbf{old}(C)$ evaluate to M and C.

The idea of this rule is that since one knows the arguments that are passed over the session event $(\rho, \textbf{old}(e), \textbf{old}(\iota_\rho(e)), \textbf{old}(state_\rho(e)), m)$, as well as the receiving actor, one can instantiate the values to get an expression that evaluates to the same value as the environment expression (the state in which the environment expression is evaluated is the same as the state when the trigger message is received).

We also can talk about the session event in the response message if the response message is associated with a protocol ρ'. For this we also permit rewriting actor service (4.4) to

$$\begin{array}{l} e.m(\overrightarrow{e_i}) \rightsquigarrow e'.m'(\overrightarrow{e_i'}) \\ \quad \textit{where } A * \varphi_{\rho'}(e', \iota_{\rho'}(e'), \textit{state}_{\rho'}(e'), m'(y; \overrightarrow{x_i}), e^*) = e^*[\overrightarrow{e_i'}/\overrightarrow{x_i}][e'/y] \end{array}$$

if the framing conditions hold.

Next, we rewrite actor service (4.3) to

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n))} \\ \textit{where} & \begin{array}{l} \mathbf{old}(\iota_{SM}(M)) = \iota_{SM}(M) * \\ C = \varphi_{SM}(M, \mathbf{old}(\iota_{SM}(M)), \mathbf{Q}, \textit{query}(c, n), c) \end{array} \end{array} \tag{4.5}$$

where we used that $\mathbf{old}(\textit{state}_{SM}(M)) = \mathbf{Q}$ from the precondition of the query trigger message. This rewrite step is supported by the original actor services logic. To continue we need to specify an actor service that describes the reaction of the manager when receiving the result message. This can be done by deriving the following local actor service

$$\begin{array}{l} \underline{M}.\mathsf{result}(\underline{r}) \rightsquigarrow \exists C'.\ C'.\mathsf{sol}(r) \\ \textit{where } C' = \varphi_{SM}(M, \mathbf{old}(\iota_{SM}(M)), \mathbf{Q}, \textit{query}(c, n), c) \end{array} \tag{4.6}$$

The derivation requires using the protocol invariant which gives the value of the M.client field in the $\mathbf{R}$ state in terms of the appropriate environment expression. The where-clause is framed since the correct session predicate is held in the precondition of the result trigger message handler. Note that this actor service does not mention M.client. So, the environment expression allows us to hide implementation details in this case. This is one advantage over the approaches in Section 3. If the manager, for example, changes the client field name, then the actor service will still hold.

We can now compose actor service (4.5) with an instantiation of actor service (4.6) to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n))$$

which is exactly the desired actor service. We used that the session identifier in the trigger and response states are the same in actor service (4.5). Using this one can then derive that the environment expression specifying the client in the query trigger message in actor service (4.5) is the same as the environment expression specifying the existentially quantified client in the sol result message in actor service (4.6).

We have derived the desired actor service (4.2) without ever mentioning anything about the client stored in the manager's local state and without the worker actor having to know anything about this client. In actor service (4.6) this client showed up for the first time in terms of an environment expression (but without mentioning the manager's local state). We could then, using the newly introduced rewrite rule, recover the connection between C in the query trigger message and the existentially quantified client in the sol response message. This is fundamentally different to all the approaches in Section 3. In those approaches we had to specify that $M.\mathsf{client} = C$ in the first local actor service describing the manager's reaction to the query message. The reason is that in those approaches the only way to talk about the client specified in the query message is inside the query message handler. Using environment expressions we can now talk about that particular client at any point in the session after the query was received. As we saw in this section, this reconstruction facilitates modular reasoning.

4.6 An example involving subsessions

Until now we have mainly considered the example given in Figure 4. Only one of the actors (the manager) in the example changes its behaviour according to a protocol. Let us now consider a modification of this actor program where the client and manager have the same implementations and the worker actor is instead given by Figure 5. The message sequence chart for the complete program is given in Figure 6.

In this implementation the worker delegates the work to a subworker which sends the correct result back. Here the worker is running a protocol that is similar to the protocol run by the manager. Note that even though we changed the worker implementation the overall response property stays the same, since eventually the worker sends the same result to the manager.

Recall that we briefly considered such a modification in Section 3.2, and noticed that in the original logic it did not seem possible to prove the desired response property modularly. In particular, it seemed that the subworker needed to know about the manager, which is clearly not modular. Next, we show that we can verify the desired response property modularly using the newly introduced techniques.

For the worker we include the exclusive permission to the subw field in the actor invariant and we define a protocol description SW, for which the protocol states are given by **CW** (where compute messages are received) and

```
actor Worker {
    Manager manager;
    Subworker subw;
    WState wstate;

    Worker() {
        this.wstate := ACCEPTCOMPUTE;
    }

    handler compute(Manager m, int n) {
        if(this.wstate == ACCEPTCOMPUTE) {
            this.manager := m;
            this.wstate := WAITFORWRESULT;
            this.subw.scompute(this, n);
        } else {
            fail();
        }
    }

    handler wresult(int n) {
        if(this.wstate == WAITFORWRESULT) {
            this.wstate := ACCEPTCOMPUTE;
            this.manager.result(n);
        } else {
            fail()
        }
    }
}

actor Subworker {
    handler scompute(Worker w, int n) {
        w.wresult(f(n))
    }
}
```

Figure 5: ActorPL program describing a worker actor that delegates work to a subworker, which gives back the result. The worker communicates with the manager given in Figure 4 (and replaces the worker in that figure).

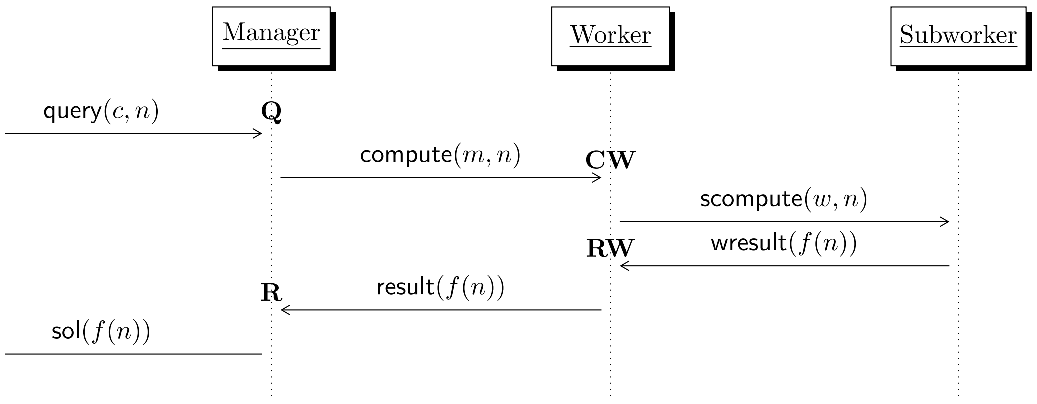

Figure 6: Message sequence chart showing an interaction involving subsessions. For each message received by the manager and the worker, the expected session state is annotated.

RW (where wresult messages are received). The transition relation is given by $\mathbf{CW} \sqsubset_{SW} \mathbf{RW}$. To ensure the messages can only be sent to the worker in the correct states and the worker can send the result message to the manager, we define the precondition of the worker's compute message handler as

$$SW(\textsf{this}) * state_{SW}(\textsf{this}) = \mathbf{CW} * SM(m) * state_{SM}(m) = \mathbf{R}$$

and we define the precondition of the worker's wresult message handler as

$$SW(\textsf{this}) * state_{SW}(\textsf{this}) = \mathbf{RW}$$

We define the precondition of the subworker's scompute message handler as

$$SW(\textsf{w}) * state_{SW}(w) = \mathbf{RW}$$

so that the subworker gets permission to send the wresult message. Furthermore, we spawn a session for SW in session state **CW** in the worker's constructor. We provide the following constructor postcondition

$$SW(\textsf{this}) * state_{SW}(\textsf{this}) = \mathbf{CW}$$

For the manager we assume the same preconditions and protocol invariant as introduced in Section 4.5. The manager gets hold of the session predicates required for sending the compute message when spawning the worker (over the constructor postcondition). We define the protocol invariant for SW as

$$\begin{aligned}&\mathit{Inv}_{SW}(s) := \textsf{\textbf{acc}}(\textsf{\textbf{this}}.\textsf{wstate}) * \textsf{\textbf{acc}}(\textsf{\textbf{this}}.\textsf{manager}) * \\ &(s = \mathbf{CW} \Rightarrow \mathit{this.state} = \textsf{ACCEPTCOMPUTE}) * \\ &\left(\begin{array}{l} s = \mathbf{RW} \Rightarrow \textsf{\textbf{this}}.\textsf{state} = \textsf{WAITFORWRESULT} * \\ \textsf{\textbf{this}}.\textsf{manager} = \varphi_{SW}(\mathit{this}, \iota_{SW}(\mathit{this}), \mathbf{CW}, \mathit{compute}(m,n), m) * \\ \mathit{SM}(\textsf{\textbf{this}}.\textsf{manager}) * \mathit{state}_{\mathit{SM}}(\textsf{\textbf{this}}.\textsf{manager}) = \mathbf{R} * \\ \iota_{SM}(\textsf{\textbf{this}}.\textsf{manager}) = \varphi_{SW}(\textsf{\textbf{this}}, \iota_{SW}(\textsf{\textbf{this}}), \mathbf{CW}, \mathit{compute}(m,n), \iota_{SM}(m)) \end{array}\right)\end{aligned}$$

Some parts of this protocol invariant are similar to the manager's SM protocol invariant. For example, in the $\mathbf{RW}$ state the manager field evaluates to the manager specified in the compute message sent over the session event in the $\mathbf{CW}$ state of the same session. We specify this analogously to the manager's protocol invariant using an environment expression.

The worker receives the manager's session predicate with respect to SM over the compute message. In the previous example it directly sent the result message in the compute message handler, hence it gave the session predicate back right away. In this example, the worker first needs to send a message to the subworker and wait for its result. The worker needs to keep the manager's session predicate for this time frame. We express this by keeping the manager's session predicate in the protocol invariant along with the fact that the session state is $\mathbf{R}$ using the following assertion in the $\mathbf{RW}$ state

$$\mathit{SM}(\textsf{\textbf{this}}.\textsf{manager}) * \mathit{state}_{\mathit{SM}}(\textsf{\textbf{this}}.\textsf{manager}) = \mathbf{R}$$

Finally, we also express that, while the manager is holding this partial session ownership, the session identifier of the corresponding session is the same as when the query message was received. We express this using an environment expression with the following assertion

$$\iota_{SM}(\textsf{\textbf{this}}.\textsf{manager}) = \varphi_{SM}(\mathit{this}, \iota_{SW}(\textsf{\textbf{this}}), \mathbf{CW}, \mathit{compute}(m,n), \iota_{SM}(m))$$

This assertion is framed by the protocol invariant. The protocol invariant contains permission to **this**.manager. Furthermore, $\iota_{SM}(m)$ is framed by the precondition of compute(m,n), since it contains the corresponding session predicate. $\iota_{SM}(\mathit{this.manager})$ is framed by $\mathit{SM}(\mathit{this.manager})$. $\iota_{SW}(\mathit{this})$ is framed by default in the SW protocol invariant.

Using these specifications it should be clear that all the message handlers are valid. We can derive the following local actor service for the worker (where the existentially quantified variable S is a subworker)

$$\underline{W}.\textsf{compute}(\underline{M}, \underline{n}) \rightsquigarrow \exists S.\ S.\textsf{scompute}(W, n)\ \ \mathit{where}\ \textsf{\textbf{old}}(\iota_{SW}(W)) = \iota_{SW}(W)$$

Furthermore, we can derive the following local actor service for the subworker

$$\underline{S}.\mathsf{scompute}(\underline{W}, \underline{n}) \rightsquigarrow W.\mathsf{wresult}(f(n)) \;\; \textit{where } \textbf{old}(\iota_{SW}(W)) = \iota_{SW}(W)$$

We can compose the two local actor services to get

$$\underline{W}.\mathsf{compute}(\underline{M}, \underline{n}) \rightsquigarrow W.\mathsf{wresult}(f(n)) \;\; \textit{where } \textbf{old}(\iota_{SW}(W)) = \iota_{SW}(W)$$

Analogously to Section 4.3 we can rewrite this actor service to

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{W}.\mathsf{compute}(\underline{M}, \underline{n}) \rightsquigarrow W.\mathsf{wresult}(f(n))} \\ & \textbf{old}(\iota_{SW}(W)) = \iota_{SW}(W) * \\ \textit{where} & M = \varphi_{SW}(W, \textbf{old}(\iota_{SW}(W)), \mathbf{CW}, \textit{compute}(m, n), m) \\ & \textbf{old}(\iota_{SM}(M)) = \varphi_{SW}(W, \textbf{old}(\iota_{SW}(W)), \mathbf{CW}, \textit{compute}(m, n), \iota_{SM}(m)) \end{array}$$

In this step, we are relating the manager given in the compute message, as well as the manager's session identifier at the time when the compute message is received with the environment of the worker's session. This way we will be able to connect the knowledge of the worker (which is in terms of its own environment) with these values.

Next, we can derive the following local actor service for the worker

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{W}.\mathsf{wresult}(\underline{n}) \rightsquigarrow \exists M'.\ M'.\mathsf{result}(n)} \\ \textit{where} & \begin{array}{l} M' = \varphi_{SW}(W, \textbf{old}(\iota_{SW}(W)), \mathbf{CW}, \textit{compute}(m, n), m) \\ \iota_{SM}(M') = \varphi_{SW}(W, \textbf{old}(\iota_{SW}(W)), \mathbf{CW}, \textit{compute}(m, n), \iota_{SM}(m)) \end{array} \end{array}$$

To verify the where-clause one needs to use the worker's protocol invariant which, for example, states what value its manager field has in terms of its own session environment. Next, we compose the previous actor service with an instantation of this local actor service to get

$$\underline{W}.\mathsf{compute}(\underline{M}, \underline{n}) \rightsquigarrow M.\mathsf{result}(f(n)) \;\; \textit{where } \textbf{old}(\iota_{SM}(M)) = \iota_{SM}(M)$$

The reasoning is analogous to the final composition in Section 4.3. The only difference is that we also reason about the environment expression involving the manager's session identifier with respect to *SM* in the different actor services. Here one can see why it was important to include the information about the manager's session identifier with respect to *SM* in the worker's protocol invariant. If we had not included this information then we would not know which session the result message is associated with.

The derived actor service is identical to the local actor service derived for the worker in the simpler program given in Figure 4. So from the actor service view even though we have modified the implementation, the behaviour of the worker is described exactly the same way. Hence the remaining derivation to get to the desired actor service (M1) is identical as in Section 4.3. This shows one modularity feature of our approach. In the modified program, for the manager only the local actor service where the manager sends the `compute` message needs to be re-verified (because the worker's specification has changed), but the rest remains unaffected. Another important point about the specification is that the subworker does not need to know anything about the manager, which would not be the case using the approach given in Section 3.2.

Suppose we change the subworker implementation to delegate to another subworker: then, the manager would notice nothing about this change. The worker would have to make sure it respects the modified interface of the subworker (essentially taking on the role of the manager in the previous example). In general, the approach introduced up to this point works well in such cases where an actor a delegates work to another actor a^* and eventually gets the expected result back, at which point the session of a^* finishes. In some sense one can regard a^*'s session as a *subsession* of a's session because it is only active between getting an initial message from a and sending the next message to a. In the next section we consider more general scenarios.

4.7 Beyond subsessions

Consider the actor program that consists of the customer actor given in Figure 7, the mediator actor given in Figure 8 and the operator actor given in Figure 9. We refer to this program (which may consist of other actors as well) as the *CMO actor program*. In this program the customer communicates with the mediator and the mediator communicates with the operator. The customer and the operator never communicate with each other directly. To get a better overview of the communication consider the message sequence chart given in Figure 10. Each of these actors changes their behaviour according to a protocol. We can express a single run of the protocol as a session. We call the corresponding protocol descriptions SC for the customer, SME for the mediator and SO for the operator. We define their transition relations as follows.

$$\mathbf{IC} \sqsubset_{SC} \mathbf{AC} \sqsubset_{SC} \mathbf{RC}$$

$$\mathbf{QM} \sqsubset_{SME} \mathbf{DM} \sqsubset_{SME} \mathbf{GM} \sqsubset_{SME} \mathbf{RM}$$

$$\mathbf{CO} \sqsubset_{SO} \mathbf{GO}$$

```
actor Customer {
    Mediator md;
    Cstate cstate;

    Customer() { this.cstate := INIT; }

    handler init(Mediator mediator, int n)
    {
        if(this.cstate == INIT) {
            this.md := mediator;
            this.cstate := WAITFORADVANCE;
            mediator.query(this, n);
        } else {
            fail();
        }
    }

    handler advance(int r)
    {
        if(this.cstate == WAITFORADVANCE) {
            this.cstate := WAITFORCRESULT;
            this.md.getresult(g(r));
        } else {
            fail();
        }
    }

    handler cresult(int res) { ... }
}
```

Figure 7: ActorPL customer actor. The mediator implementation is given in Figure 8.

```
actor Mediator {
    Customer c;
    Operator op;
    Mstate mstate;

    Mediator() { this.mstate := ACCEPTQUERY; }
    handler query(Customer customer, int n)
    {
        if(this.mstate == ACCEPTQUERY) {
            this.c := customer;
            Operator operator := spawn Operator();
            this.mstate := WAITFORDONE;
            this.op := operator;
            operator.calc(this, n);
        } else { fail(); }
    }
    handler done(int n)
    {
        if(this.mstate == WAITFORDONE) {
            this.mstate := ACCEPTGETRESULT;
            this.c.advance(n);
        } else { fail(); }
    }
    handler getresult(int n)
    {
        if(this.mstate == ACCEPTGETRESULT) {
            this.mstate := WAITFORMRESULT;
            this.op.get(n);
        } else { fail(); }
    }
    handler mresult(int res)
    {
        if(this.mstate == WAITFORMRESULT) {
            this.mstate := ACCEPTQUERY;
            this.c.cresult(res);
        } else { fail(); }
    }
}
```

Figure 8: ActorPL mediator actor. The customer implementation is given in Figure 7 and the operator implementation is given in Figure 9.

```
actor Operator {
    int val;
    Mediator md;
    State state;

    Operator() { this.state := ACCEPTCALC; }
    handler calc(Mediator mediator, int n)
    {
        if(this.state == ACCEPTCALC) {
            this.md := mediator;
            this.val := n;
            this.state := ACCEPTGET;
            mediator.done(f(n));
        } else {
            fail();
        }
    }

    handler get(int n)
    {
        if(this.state == ACCEPTGET) {
            this.state := ACCEPTCALC;
            this.md.mresult(h(this.val+n));
        } else {
            fail();
        }
    }
}
```

Figure 9: ActorPL operator actor. The mediator implementation is given in Figure 8.

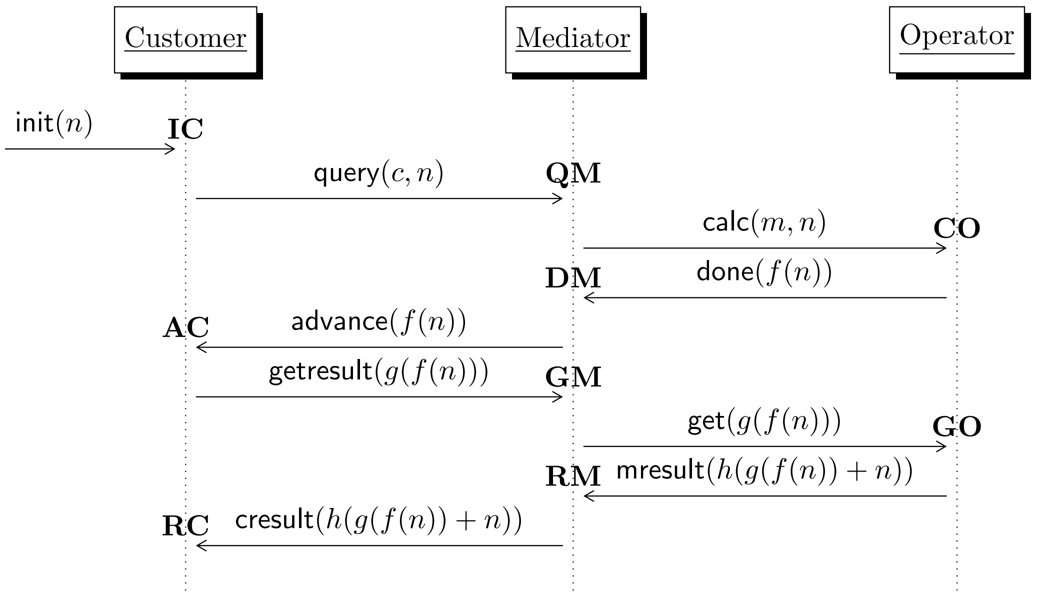

Figure 10: Message sequence chart depicting communication between the customer given in Figure 7, the mediator given in Figure 8 and the operator given in Figure 9. For each message the session state in which it is accepted by the receiving actor is annotated.

We assume the straightforward association of protocol states to messages that are expected. In the case of the customer, in **IC** the init message is expected, in **AC** the advance message is expected and in **RC** the cresult message is expected. The association for the mediator and service is analogous.

One important aspect is that, for example, the mediator sends a message to the operator and once it gets a message back, the operator's session does not end. In fact, the mediator sends another message to the operator where the operator is still in the same session as before. This is a difference to the examples we have considered up to now (see Section 4.6). Note that the mediator stores the operator in its op field without modifying the field until the session finishes. This ensures the mediator is always talking to the same service during its session. Analogous observations can be made for the customer and the mediator.

Let us investigate how we would write the preconditions of the different message handlers. Consider the operator's calc message handler. We need to put the session predicate corresponding to *SO* into it, since the message handler is associated with *SO*. Furthermore, because the operator needs to send a done message to the mediator upon receiving the calc message, the session predicate corresponding to *SME* is also required. Hence we could provide the following precondition

$$\mathit{SO}(\textbf{this}) * \mathit{state}_{\mathit{SO}}(\textbf{this}) = \mathbf{CO} * \mathit{SME}(m) * \mathit{state}_{\mathit{SME}}(m) = \mathbf{DM}$$

Next, consider the mediator's done message handler. In the precondition we need to specify the session predicate corresponding to *SME*. Furthermore, because the mediator needs to send a get message to the operator stored in its op field at some point later, we should also provide the operator's session predicate corresponding to *SO*. The only way it seems possible to specify this is as follows

$$\mathit{SME}(\textbf{this}) * \mathit{state}_{\mathit{SME}}(m) = \mathbf{DM} * \mathit{SO}(\textbf{this}.\mathsf{op}) * \mathit{state}_{\mathit{SO}}(\textbf{this}.\mathsf{op}) = \mathbf{GO}$$

There are two problems here. First, this precondition is not self-framing because **this**.op is not framed. Second, the mediator's implementation details are exposed. If we ignore the second problem, we could make this approach work by sending permission to the op field over the calc message along with the knowledge that it evaluates to the operator receiving the message. However, the mediator does not know if the session predicate that it receives over the done message is associated with the same session as when it sent the calc message (i.e. if they have the same session identifier). This fact will become important, once we derive response properties. The intuition is that one learns facts about the other sessions at different points and one needs to be able to associate the facts corresponding to the same actor and protocol with the same session. Similar problems, where actors communicate with each other over multiple messages and need to obtain the session predicates as well as the knowledge that these session predicates belong to the same session, also show up for the interaction between the customer and the mediator.

The underlying issue in our current approach is that if an actor requires a session predicate to another session at some point, then one can generally only ensure this by explicitly transferring this session predicate over some message. Hence one can only specify this session predicate at a point where the required session predicate will be in the correct session state. However, this is not sufficient for this scenario, because most of the messages do not contain information about the identities of the actors, making it hard to avoid exposing implementation details. Furthermore, one needs to associate a set of session predicates received from the same actor with the same session.

An idea for a solution is as follows. Consider the mediator and the operator. The mediator and the operator know the sequence of messages that is going to be exchanged before the interaction has even started. Hence there should be a way for them to agree beforehand over which message which session predicate will be sent to the other actor. They could also agree that all the session predicates belonging to one actor would have to belong to the same session.

They could make such an agreement when the mediator sends a query message to the operator. It would state that the operator gets hold of the session predicate for the done message and for the operator to send the done message it would have to send along its session predicate for the get message.

Such an approach would avoid having to explicitly mention all the session predicates explicitly in preconditions of the messages over which they are transferred and it would also make sure that one can associate the different session predicates with the same session.

Another way of looking at this approach is that the mediator and the operator agree on an interaction which consists of a sequence of messages that are exchanged and it is then implicitly clear which session predicates must be sent over which messages for the interaction to be executed as specified.

In the next section we develop a way for parties to make such an agreement. This will enable us to provide a modular specification for the example.

4.8 Interaction permissions and request clauses

In this section we develop a way for two parties to agree on an interaction when they meet. For this we need a way to talk about which message can be sent once a particular message is received. Therefore we introduce *send event permissions* and *receive event assertions*. A send event permission is of the form $\textsc{send}(E)$ where E is a session event. It represents the session predicate permission required to send the message associated with E. If $E = (\rho, a, i, s, m)$ then we write $\textsc{send}_\rho(a, i, s, m)$ to denote $\textsc{send}(E)$. The permission contained by (ρ, a, i, s, m) is given by

$$P_{send} := \rho(a) * state_\rho(a) = s * \iota_\rho(a) = i$$

The difference between the send event permission and P_{send} is that the send event permission cannot be used to send messages associated with ρ *other than* the m message. We permit exchanging P_{send} for the corresponding send event permission (but not vice versa, since as discussed there is a difference). However, it must be ensured that the session predicate is not under the modality introduced in Section 4.2.2. This means the send event permission has not already been used to send a message associated with the protocol in the current session state.

The reason for this distinction is that it provides a way to enforce that a particular message is sent, which in turn makes sure that an interaction proceeds as one has specified. This is particularly important if one wants to support non-deterministic interactions with branching behaviour (our protocols are deterministic, but if one wants to generalize, then this becomes important).

Since the session predicate is contained in this send event permission, the send event permission is not duplicable. Furthermore, the send event permission frames everything that the corresponding session predicate frames. Send permissions can be sent around (with the exception explained above).

A receive event assertion is of the form $\textsc{rcv}(E)$ where E is a session event. If it is held, then this means that the session event E has occurred. If $E = (\rho, a, i, s, m)$ then we write $\textsc{rcv}_\rho(a, i, s, m)$ to denote $\textsc{rcv}(E)$. For a message handler m associated with protocol definition ρ one may assume the receive event assertion $\textsc{rcv}_\rho(this, \iota_\rho(this), state_\rho(this), m)$ at the beginning of the message handler body.

In contrast to the send event permission, the receive event assertion is duplicable. The fact that a particular session event has occurred remains true.

Before we continue, we make an important observation. It is not possible for someone to own $\textsc{send}(E)$ and $\textsc{rcv}(E)$ to hold. The reason is as follows. $\textsc{rcv}(E)$ holds once session event E occurs. Assume this session event occurs when message m which is associated with the protocol is received. At this point the session is in the state specified by E, however the session predicate is under a modality (see Section 4.2.2). Hence it cannot be used to get $\textsc{send}(E)$. One can only get rid of the modality once the session is progressed at which point it is ensured that the session will never again be in the state specified by E (due to strict ordering on the session states). Hence it cannot be possible to generate $\textsc{send}(E)$ from the point where $\textsc{rcv}(E)$ holds.

Using send event permissions and receive event assertions we can now define *interaction permissions*, which describe an interaction from the point of view of one of the two *interaction endpoints*. The syntax of interaction permission I can be described by the following production rules

$$\begin{aligned} I &\to R \mid S \\ S &\to \textsc{send}(E)\ .\ R \mid \textsc{EndS} \\ R &\to \textsc{rcv}(E)\ .\ S \mid \textsc{EndR} \end{aligned}$$

where E ranges over session events. We introduce *request clauses* as part of the specification of a message handler (along with the already existing precondition) which can hold such interaction permissions. This allows actors to *request* an interaction in the request clause of a message handler. We call the actor requesting the interaction the *requestor*. The first event in the requested interaction must be a send event. Furthermore, the receive events may only be with respect to the session corresponding to the requestor. In our example, we provide the following interaction permission in the request clause of the operator's calc message handler

$$\begin{aligned} &\textsc{send}_{SME}(\mathsf{mediator}, \iota_{SME}(\mathsf{mediator}), \mathbf{DM}, \mathsf{done})\ . \\ &\textsc{rcv}_{SO}(\mathbf{this}, \iota_{SO}(\mathbf{this}), \mathbf{GO}, \mathsf{get})\ . \\ &\textsc{send}_{SME}(\mathsf{mediator}, \iota_{SME}(\mathsf{mediator}), \mathbf{RM}, \mathsf{mresult})\ .\ \textsc{EndR} \end{aligned} \tag{4.7}$$

This request describes the following interaction. The operator wants to send the done message to the mediator now, and after doing this it guarantees to be in a session state where it receives the get message. In particular, it guarantees that the other interaction point (i.e. the mediator) can get hold of the permission to send the get message once done has been received. We will see how this implicit permission transfer is achieved. Finally, once the operator has received the get message, it wants the permission to send the mresult message to the mediator, at which point the complete interaction ends. Note that the session identifier for both send event permissions are the same, so in the interaction these send events are with respect to the same session.

When an actor sends a message that contains a request clause then it must satisfy the precondition and *accept* the request. We call the actor accepting the request the *acceptor*. The acceptor must give up the send permission associated with the first event in the request clause to ensure that the requestor gets hold of this permission. Furthermore, we demand from the acceptor and requestor the following:

- The acceptor must give up one session finalization access permission for each session appearing in send events of the interaction. The reason is that one can then ensure that each of these sessions is active at this point and that they are not going to finish until the interaction is finished.
- At the beginning of the message handler for which the request is specified, the requestor must give up one session finalization access permission for the session associated with the message handler (i.e. the session specified in all receive events of the interaction).

The motivation for this will become clearer soon. Finally, we require that the request clause is framed by the precondition to ensure we only talk about well-defined expressions. Since we may think of the session finalization access permissions provided by the acceptor as part of the precondition, all expressions of the form $\iota_\rho(a)$ are framed, where $\iota_\rho(a)$ is an identifier corresponding to one of the send events in the request clause (see Section 4.2.3). Therefore, the shown request clause is framed by the precondition ($\iota_{SO}(\textbf{this})$ is framed by the session predicate $SO(\textbf{this})$ that must be provided in the precondition).

Once the message is sent (i.e. the request has been accepted), the acceptor gets hold of the *dual* of the requested interaction permission and once the requestor receives the message it obtains the requested interaction permission. We call these two interaction permissions originating from the same request as *companion permissions* (i.e. we say the companion permission of I to refer to the other permission). The main idea, as we will see, is that one can implicitly transfer send event permissions between these companion permissions. The dual $\mathcal{D}(I)$ of interaction permission I is defined as follows:

$$\begin{aligned} \mathcal{D}(\textsc{send}(E) \,.\, R) &= \textsc{rcv}(E) \,.\, \mathcal{D}(R) \\ \mathcal{D}(\textsc{rcv}(E) \,.\, S) &= \textsc{send}(E) \,.\, \mathcal{D}(S) \\ \mathcal{D}(\textsc{EndS}) &= \textsc{EndR} \\ \mathcal{D}(\textsc{EndR}) &= \textsc{EndS} \end{aligned}$$

Hence send event permissions are exchanged with receive events and vice versa. The dual is self inverse. In our example, the mediator gets hold of the dual of (4.7) which is given by (we replaced the mediator and service references with expressions from the mediator's point of view)

$$\begin{aligned}&\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{DM}, \textsf{done}) \,.\\ &\textsc{send}_{SO}(\textsf{\textbf{this}}.\textsf{op}, \iota_{SO}(\textsf{\textbf{this}}.\textsf{op}), \mathbf{GO}, \textsf{get}) \,.\\ &\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{RM}, \textsf{mresult}) \,.\, \textsc{EndS}\end{aligned} \tag{4.8}$$

Right after the request message has been accepted, the session finalization access permissions given up by the acceptor for all the sessions occurring in send events are owned jointly by both companion permissions (the permission for the requestor is only explicitly obtained once the message is received, but we can still think of this permission as being generated once the message is sent). The session finalization access permission for the receive events is only given up once the message is received by the requestor, at which point it is also jointly owned by the companion permissions. However, because we provide the session predicate associated with the receive events when sending the message, we can be sure that this session is not going to be finished before the message is received (this is one reason for the restriction on the receive events in the request). As we will see later in this section (and show in detail in Section 4.9), we guarantee that one cannot retrieve the session finalization permission from the interaction permission as long as the session is specified in a session event in either of the two interaction permissions (i.e. as long as the session is *active* in either of the two permissions). This means as long as a session is active in an interaction permission it cannot finish. Therefore an interaction permission frames all expressions of the form $\iota_\rho(a)$ where this is a session identifier corresponding to an active session. We can therefore conclude that the interaction permissions (4.7) and (4.8) are framed in any program state as long as permission to the heap-dependent expressions is held.

Next, we describe how such an interaction permission can be used to progress the interaction. If the first event of the interaction permission is a send event, then the corresponding send event permission is held by that interaction permission (that's why the acceptor must give up a send event permission). However, to be able to use this send event permission (and hence advance the interaction) one must give up the send event permission for the next receive event in the interaction. We can express this using the following entailment

$$\textsc{send}(E') * (\textsc{send}(E) \,.\, \textsc{rcv}(E') \,.\, S) \models \textsc{send}(E) * (\textsc{rcv}(E') \,.\, S)$$

As one can see, one ends up with the send event permission and the continuation of the interaction permission. The send permission that is given up is implicitly transferred to the companion interaction permission, from where one will be able to retrieve it using a receive event.

Once the operator has received the calc message, it gets hold of interaction permission (4.7), hence by giving up

$$\textsc{send}_{SO}(\textsf{\textbf{this}}, \iota_{SO}(\textsf{\textbf{this}}), \mathbf{GO}, \textsf{get})$$

it gets hold of the send permission for the done message and the continuation. A special case is if the next receive event is the end of the interaction, in that case nothing needs to be given up to get the send permission. We can express this using the following entailment

$$\textsc{send}(E) \,.\, \textsc{EndR} \models \textsc{send}(E)$$

If the first event of the interaction permission is a receive event then one may use the corresponding receive event assertion to get hold of the continuation. We have

$$\textsc{rcv}(E) * (\textsc{rcv}(E) \,.\, S) \models S$$

Once the mediator has sent the query message and accepted the request, it gets hold of interaction permission (4.8). Hence once it receives the done message, it may use the corresponding receive event assertion to get hold of the continuation which contains the send permission for the get message (however, as before it cannot use this without giving up send event permission for the next receive event).

The intuition why the send permission to get can be claimed is the following. When the operator gives up the send event permission for the get message to advance its interaction, then this permission is implicitly transferred to the companion permission held by the mediator. In Section 4.9 we will give an argument which shows the following. If the send event permission is given up to advance the interaction permission, then only the entity owning the companion permission can get hold of this send event permission (using the corresponding receive event assertion).

Additionally to getting hold of the continuation of the interaction permission using the receive event, one also gets hold of the session finalization access permission associated with the session specified by the receive event, if one can guarantee that the corresponding session is not active in the continuation. Hence we can make the entailment describing the progression of a interaction permission where the first event is a receive event more precise. A session is uniquely identified by a protocol description ρ, an actor a and a session identifier i. Let $\mathit{inactive}(I, (\rho, a, i))$ evaluate to true if the session is not contained in any session event in I (i.e. it is *inactive* in I). In our example this check is syntactic because each protocol only appears as part of a single session in the interaction permission (later this will change and we'll mention how to do the check then). We have

$$\textsc{rcv}_\rho(a,i,s,m) * (\textsc{rcv}_\rho(a,i,s,m) \,.\, S) \models \\ S * (\mathit{inactive}(I,(\rho,a,i)) \Rightarrow \rho^F(a,1^+))$$

The intuition why we can claim the session finalization permission is the following. As we said earlier, the two companion permissions jointly own a session finalization access permission for each session that is active in either of the two interaction permissions. Hence $\rho^F(a,1^+)$ is owned jointly by the two companion permissions as well, since the specified session is active in one of the two interaction permissions. We can identify the last event involving the session in the interaction. In one of the companion permissions this will be a send event, in the other it will be a receive event. One can show that if one of the permissions ever progresses to a point where the next event in the interaction is the last receive event, then it must hold that if this receive event occurs, the other interaction permission has gone past the corresponding send event (because the send event in a sense enables the receive event). Therefore, after the receive event is applied to get the continuation, neither of the two companion permissions mention the session, hence it is fine for the finalization permission to be extracted.

In our example, the mediator can get hold of its finalization permission if it has progressed its interaction permission to

$$\textsc{rcv}_{SME}(\textbf{this}, \iota_{SME}(\textbf{this}), \mathbf{RM}, \mathsf{mresult}) \,.\, \textsc{EndS}$$

and has received the mresult message.

4.9 Soundness argument for interaction permissions

Even though we have not defined the formal semantics of interaction permissions, we give an argument for why it is sound to use them in the way we have described. In Section 4.8 we showed the different ways one can manipulate interaction permissions (by progressing them) using entailments. A more precise way of thinking about this is that each of the shown entailments corresponds to *repartitioning* the permissions held in the interaction permission. As we saw there are three repartitioning steps:

1. **Accept step.** If a message is sent which contains a request clause with interaction permission $I := (\textsc{send}(E) \,.\, R)$ and the acceptor satisfies all the conditions (i.e. gives up $\textsc{send}(E)$ and a session finalization

access permission for each session specified in a send event in I), then interaction permission ($\textsc{send}(E) \,.\, R$) and ($\textsc{rcv}(E) \,.\, S$) are created.

2. **Send step.** Given ($\textsc{send}(E) \,.\, R$) and the send event permission corresponding to the first receive event in R (which is empty if $R = \textsc{End}\mathrm{R}$), then repartition these permissions to ($\textsc{send}(E) * R$).

3. **Receive step.** Given ($\textsc{rcv}(E) \,.\, S$) and $\textsc{rcv}(E)$, then repartition the permission to S and if the session corresponding to E is inactive in S, also extract the corresponding session finalization access permission.

We need to guarantee that it is sound to obtain the send event permission that is obtained in the send step and the session finalization access permission that is obtained in the receive step. In particular, this means that the session specified by the send event is active and no one else has the send event permission, otherwise interaction permissions would enable unsound reasoning. For example, suppose one could get hold of $\textsc{send}(E)$ even though someone else already owns a $\textsc{send}(E)$ simultaneously. Then this means that we have permission to send a message m associated with the protocol in a session state, even though the session is not in that session state or may not even be active. This would be unsound because one may assume the protocol invariant for that session state at the beginning of message handler m even though the session is not in that state.

If one could get hold of a session finalization access permission that did not originate from the session finalization source permission associated with the corresponding session, then one can also show that this would lead to unsound reasoning (one possible argument involves deriving an actor service that does not hold).

The main idea is that we track the interaction permissions as they progress. For this we assume that each interaction permission I that is generated in the accept step obtains a unique label l. When an interaction permission I with label l is progressed using the send or the receive step, then after the step the continuation permission gets the same label l (hence the label l remains the same before and after the step, but the interaction permission itself changes). This labelling allows us to identify for each interaction permission the companion interaction permission using its label. Two companion permissions have different labels, but there is a function mapping a label to its companion's label.

To ensure that one may obtain the permissions using the repartitioning steps, we use a notion of ownership that is associated with interaction permissions. For example, as we will see, we will make sure that if an interaction permission I is of the form ($\textsc{send}(E) \,.\, R$), then the permission $\textsc{send}(E)$ *is* *owned by I*, i.e. no one else has this permission at the moment. That means at some point $\textsc{send}(E)$ must have been generated soundly (for example, after a session was progressed) and then given up in a way such that only I has access to it.

The idea of the soundness argument is to track properties for each interaction permission (some of which state what the interaction permissions own at different points), which will help us show that whenever one extracts the permissions in the receive and send steps, then these permissions are owned by the corresponding interaction permissons. Next, we present these properties and briefly give an intuition for why they should hold (later we will show that they are established and preserved).

Property 1. *There is a companion map $\mathcal{C}$ which maps labels to the corresponding companions, such that an interaction permission with label l has the unique companion interaction permission with label $\mathcal{C}(l)$ and the companion of interaction permission with label $\mathcal{C}(l)$ is l (hence $\mathcal{C}$ is self-inverse). Once a companion is set for a label, then it never changes.*

Recall that a pair of interaction permissions are generated when an interaction request is accepted. The requestor gets the interaction permission specified in the request with label l and the acceptor gets the dual with label l_c. The interaction permissions with labels l and l_c are always companions.

Property 2. *If an interaction permission I of the form $(\textsc{send}(E)\ .\ R)$ is held, then the permission associated with $\textsc{send}(E)$ is owned by I.*

If this were not the case, then progressing this interaction would be unsound.

Property 3. *If an interaction permission I is of the form $(\textsc{send}(E)\ .\ R)$, then its companion permission is of the form $\mathcal{D}(I)$.*

Property 4. *If an interaction permission I is of the form $(\textsc{rcv}(E)\ .\ S)$ and its companion is I', then one of the following holds*

- *I' is of the form $\mathcal{D}(I)$.*
- *I' is of the form $(\textsc{rcv}(E'')\ .\ \mathcal{D}(I))$.*
- *I' is of the form $\mathcal{D}(S)$ and if S is of the form $(\textsc{send}(E')\ .\ R)$; in this case the permission associated with $\textsc{send}(E')$ is owned by I.*

The general intuition for properties 3 and 4 is the following. Companion permissions describe the two interaction endpoints of a single interaction. Hence companion permissions progress in lockstep. It is not possible for one interaction permission to progress two steps while the companion does not progress at all. They always progress alternatively.

If one of the interaction permissions specifies a send event next, then we can be sure that the companion must be ready to receive this send event. If one of the interaction permission I specifies a receive event next, then there are multiple possibilities for the companion I'.

- The first possibility is that I and I' are at the same point in the interaction, this means I' is just the dual of I.
- The second possibility is that I is one step ahead of I'. This means the permission to $\textsc{send}(E'')$ was extracted and I' is still waiting to be progressed. This means I' is still waiting for the message associated with E'' to be received[10].
- The third possibility is that I' is one step ahead of I, hence $\textsc{send}(E)$ was extracted and I is still waiting to be progressed. If $S = \textsc{send}(E').R$ ($I = \textsc{rcv}(E)\ .\ S$), then to extract $\textsc{send}(E)$ the permission $\textsc{send}(E')$ must have been given up. We associate this permission at this point with I.

Property 5. *For each interaction permission I and its companion permission I', we have that the following holds. Suppose I and I' were obtained directly from the accept step and let $\mathcal{S}$ be the session associated with the request clause in the corresponding accept step. We have that for every session except $\mathcal{S}$ that is active in I or I' a session finalization access permission is owned jointly by I and I'. For $\mathcal{S}$ we have that the session is currently active and will remain active until I and I' are first progressed, at which point a corresponding session finalization access permission for $\mathcal{S}$ is owned jointly by I and I'.*

Suppose I or I' were not obtained from the accept step (i.e. one of them was obtained after a receive or send step), then it is guaranteed that for every session that is active in I or I' a session finalization access permission is owned jointly by I and I'.

[10]This explanation gives an intuition but is not a completely correct explanation. It is possible for the message associated with E'' to be received, but the interaction permission to not be progressed. We do not force the progression of interaction permission upon reception of messages.

The reason we require these finalization permission to be owned "jointly" is that we need them to frame expressions in both companion permissions. One can think of this as one half of the finalization permission being owned by one of the interaction permissions and the other half being owned by the companion interaction permission.

We need this to show that a session cannot finish as long as it is active in an interaction permission. Note that once Property 5 is established for newly generated interaction permissions, one may assume that interaction permissions I and I' jointly own a session finalization access permission for each session that is active in I or I' whenever a send step or receive step is applied to I or I'.

Next, we show that these properties are established when interaction permissions are generated in the accept step. We then show that under the assumption of these properties, that whenever one applies a send step or a receive step then the permission that one extracts is owned by the interaction permission and after the repartitioning step the properties are preserved.

Accept step: In this case an acceptor sends a message m to a requestor, where m specifies a request clause with interaction permission $(\textsc{send}(E) . R)$. To perform this step the acceptor gives up send permission $\textsc{send}(E)$. After the step interaction permissions $I' := (\textsc{rcv}(E) . \mathcal{D}(R))$ with label l' (which the acceptor gets hold of) and $I := (\textsc{send}(E) . R)$ with label l (which the requestor gets hold of, once it receives m) are created. We define the companion of l to be l' and vice versa. Hence Property 1 is established. Properties 3 and 4 are a direct consequence of the companion permissions being duals of each other. We associate the permission $\textsc{send}(E)$, that was given up, with I. Hence Property 2 is established.

The acceptor gives up one session finalization access permission for each session that is active in I and I', except the session $\mathcal{S}$ that is associated with the request. $\mathcal{S}$ is guaranteed to be active until m is received since the corresponding session predicate is transferred over m. It is guaranteed that I' cannot progress until I progresses, since otherwise $\textsc{rcv}(E)$ holds and I owns $\textsc{send}(E)$ simultaneously which as explained at the beginning of Section 4.8 cannot hold. Furthermore, $\mathcal{S}$ cannot progress at all until message m is received by the requestor at which point the requestor must directly give up one session finalization access permission for $\mathcal{S}$ at the beginning of the message handler. We conclude that Property 5 is established.

Send step: In this case an interaction permission I with label l of the form $(\textsc{send}(E) . R)$ is progressed to R. We know from Property 3 that the companion permission I' is of the form $(\textsc{rcv}(E) \mathbin{.} \mathcal{D}(R))$.

Suppose $R = (\textsc{rcv}(E') \mathbin{.} S)$. Hence to progress I $\textsc{send}(E')$ is given up and $\textsc{send}(E)$ is extracted from I. Due to Property 2 we know that $\textsc{send}(E)$ is owned by I before the progression, hence $\textsc{send}(E)$ can be safely extracted.

Furthermore, we associate the permission $\textsc{send}(E')$, that was given up, with I' (so it is now owned by I'). This corresponds to an implicit transfer of the send event permission to the companion. After the progression the interaction permission with label l (it is given by R) is one step ahead of its companion I' and hence we conclude (since we associated $\textsc{send}(E')$ with I') that Property 4 is preserved. Properties 2 and 3 are preserved trivially because after the progression the first event in I is a receive event.

Next, suppose $R = \textsc{End}\mathrm{R}$, then the permission of $\textsc{send}(E)$ is obtained without giving anything up anything, but this is fine since extracting the send permission can be done as it was owned by I and I' is of the form $(\textsc{rcv}(E) \mathbin{.} \textsc{End}\mathrm{S})$.

Property 5 is preserved since no finalization permission is extracted.

Receive step: In this case it is known that $\textsc{rcv}(E)$ holds and interaction permission I with label l of the form $(\textsc{rcv}(E) \mathbin{.} S)$ is progressed to S. Also after the step is finished, the session finalization access permission associated with E is extracted if the corresponding session $\mathcal{S}_1$ is not active in S. We know from Property 4 that there are three possibilities for companion permission I' of I before the progression.

The first possibility is that I' is of the form $(\textsc{send}(E) \mathbin{.} \mathcal{D}(S))$. From Property 2 we know that I' owns $\textsc{send}(E)$. However, as explained at the beginning of Section 4.8, it is not possible for $\textsc{rcv}(E)$ to hold and someone to own $\textsc{send}(E)$. We conclude that I' cannot be of the suggested form.

The second possibility is that I' is of the form $(\textsc{rcv}(E'') \mathbin{.} \mathcal{D}(I))$. From Property 4 and the fact that the companion of I' is I (due to Property 1) we conclude that I' must own $\textsc{send}(E)$. However, as in the previous case this is impossible. We conclude that I' cannot be of the suggested form.

The third possibility is that I' is of the form $\mathcal{D}(S)$. After the progression the interaction permission with label l and its companion I' are duals of each other, hence Properties 3 and 4 hold. Suppose $S = (\textsc{send}(E') \mathbin{.} R)$. From Property 4 we conclude that I owns $\textsc{send}(E')$. Hence after the progression we transfer $\textsc{send}(E')$ to S (which is the interaction permission with label l). Therefore Property 2 is preserved. Next, suppose $\mathcal{S}_1$ is not active in S. Hence it is not active in $\mathcal{D}(S)$ and therefore also not in I' (since $I' = \mathcal{D}(S)$). Due to Property 5 we know that session finalization access permission for $\mathcal{S}_1$ is owned jointly by I and I' before the progression, hence we can extract it safely. Property 5 can be preserved because $\mathcal{S}_1$ is not active in S or I'.

We have shown using introduced properties that the permissions that are extracted by the receive and send steps are always owned by the interaction permission, which means that the repartitioning steps do not forge any of the permissions that are extracted.

4.10 Proving the CMO example

Consider the CMO actor program which we introduced in Section 4.7. The main response property in this program can be expressed by the following actor service

$$\underline{C}.\mathsf{init}(n_i) \rightsquigarrow C.\mathsf{cresult}(h(g(f(n_i)) + n_i)) \textit{ where } \textbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \quad (4.9)$$

This can be observed by looking at the message chart for the CMO program in Figure 10 on page 58. In this section, we show to how specify the actor program and verify this actor service in a modular fashion using the introduced interaction permissions and request clauses in Section 4.8. We assume the partial protocol definitions (protocol states and transition relations) provided in Section 4.7 for the customer protocol SC, the mediator protocol SME and the operator protocol SO.

4.10.1 Operator specification

We provide the following constructor postcondition for the operator

$$SO(\textbf{this}) * state_{SO}(\textbf{this}) = \mathbf{CO}$$

which can be established by spawning the session in the constructor. The precondition of the calc and the get message handlers just contain the operator's session predicate $SO(\textbf{this})$ in the session states $\mathbf{CO}$ (for calc) and $\mathbf{GO}$ (for get). We define the request clause of the calc message handler using the interaction permission (4.7). As discussed in Section 4.8, this interaction permission is framed by the precondition. We define the following protocol invariant for protocol description SO.

$$\begin{aligned}
Inv_{SO}(s) := {} & \textbf{acc}(\textbf{this}.\mathsf{state}) * \textbf{acc}(\textbf{this}.\mathsf{md}) * \textbf{acc}(\textbf{this}.\mathsf{val}) * \\
& s = \mathbf{CO} \Rightarrow \textbf{this}.\mathsf{state} = \mathsf{ACCEPTCALC} * SO^{F}(\textbf{this}, 0^{-}) * \\
& s = \mathbf{GO} \Rightarrow \\
& \left(\begin{array}{l}
\textbf{this}.\mathsf{state} = \mathsf{ACCEPTGET} * SO^{F}(\textbf{this}, 1^{-}) * \\
\textbf{this}.\mathsf{md} = \varphi_{SO}(\textbf{this}, \mathbf{CO}, \iota_{SO}(\textbf{this}), \mathsf{calc(m,n)}, \mathsf{m}) * \\
\left(\begin{array}{l} \iota_{SME}(\textbf{this}.\mathsf{md}) \\ = \varphi_{SO}(\textbf{this}, \mathbf{CO}, \iota_{SO}(\textbf{this}), \mathsf{calc(m,n)}, \iota_{SME}(\mathsf{m})) \end{array} \right) * \\
\textbf{this}.\mathsf{val} = \varphi_{SO}(\textbf{this}, \mathbf{CO}, \iota_{SO}(\textbf{this}), \mathsf{calc(m,n)}, \mathsf{n}) * \\
\left(\begin{array}{l}
\textsc{rcv}_{SO}(\textbf{this}, \iota_{SO}(\textbf{this}), \mathbf{GO}, \mathsf{get}) \ . \\
\textsc{send}_{SME}(\textbf{this}.\mathsf{md}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathbf{RM}, \mathsf{mresult}) \ . \\
\textsc{EndR}
\end{array} \right)
\end{array} \right)
\end{aligned}$$

The idea of the protocol invariant is that at first when the operator is in session state **CO**, it does not track any information yet that is required for the interaction with the mediator since the interaction has not started. Once the operator moves to state **GO** the interaction has started and the protocol invariant relates its local state to its session environment. For example, it tracks that the mediator which it received over the calc message is stored in its **this**.val field. Furthermore, the interaction permission that will be required to obtain the permission to send the mresult message to the mediator once the get message is received is held. Finally, since the operator made a request, one of its session finalization access permissions had to be given up. This is reflected by the assertion $SO^{F}(\textbf{this}, 1^{-})$.

This protocol invariant is self-framing, partly because the interaction permission held in state **GO** frames $\iota_{SME}(\textbf{this}.\mathsf{md})$, as the corresponding session is active in the interaction. Together with the preconditions it guarantees that only the non-failing branches are taken in the message handlers.

Consider the calc message handler body. At the beginning the operator must give up one session finalization access permission due to the request clause. It can progress its session to session state **GO** and then it can turn its session predicate into the corresponding send event permission, i.e.

$$SO(\textbf{this}) * state_{SO}(\textbf{this}) = \mathbf{GO} \models \textsc{send}_{SO}(\textbf{this}, \mathbf{GO}, \iota_{SO}(\textbf{this}), \mathsf{get})$$

the operator gets hold of the interaction permission (4.7) which is specified in the request clause at the beginning. Using the above send event permission it can advance the interaction permission, after which the operator obtains the following send event permission

$$\textsc{send}_{SME}(\textbf{this}.\mathsf{md}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathbf{DM}, \mathsf{done})$$

which the operator can use to send the done message to the mediator (the precondition only requires the session predicate in state **DM** which the send permission provides). Advancing the interaction permission also results in the continuation permission which is given by

$$\begin{aligned}&\textsc{rcv}_{SO}(\textbf{this}, \iota_{SO}(\textbf{this}), \mathbf{GO}, \mathsf{get})\ .\\ &\textsc{send}_{SME}(\textbf{this}.\mathsf{md}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathbf{RM}, \mathsf{mresult})\ .\ \textsc{EndR}\end{aligned}$$

Using this interaction permission it can satisfy part of the protocol invariant in state **GO**. We know at the beginning of the message handler that no finalization permission was given away (due to the protocol invariant), hence after giving one of the access permissions away we are left with $SO^F(\textbf{this}, 1^-)$. Finally, the assertions in the protocol invariant which involve environment expressions can be verified using the rule introduced in Section 4.4. Hence the protocol invariant can be established at the end of the message handler. We conclude that calc is valid.

Next consider the get message handler body. The corresponding receive event assertion can be used to advance the interaction permission stored in the protocol invariant. Since the operator's session is not active in the continuation it gets hold of $SO^F(\textbf{this}, 1^+)$, which means the operator again has the full finalization permission and therefore can finish the session. Furthermore, it gets hold of the send event permission required to send the mresult message. Hence get is valid.

4.10.2 Mediator specification

The preconditions of the query, done, getresult, mresult message handlers just contain the mediator's session predicate $SME(\textbf{this})$ in the session states **QM** (for query), **DM** (for done), **GM** (for getresult) and **RM** (for mresult). For the query message handler we provide a request clause with the following interaction permission

$$\begin{aligned}&\textsc{send}_{SC}(\mathsf{customer}, \iota_{SC}(\mathsf{customer}), \mathbf{AC}, \mathsf{advance})\ .\\ &\textsc{rcv}_{SME}(\textbf{this}, \iota_{SME}(\textbf{this}), \mathbf{GM}, \mathsf{getresult})\ .\\ &\textsc{send}_{SC}(\mathsf{customer}, \iota_{SC}(\mathsf{customer}), \mathbf{RC}, \mathsf{cresult})\ .\ \textsc{EndR}\end{aligned} \tag{4.10}$$

It is framed by the precondition. It describes the interaction the mediator expects with the customer. The protocol invariant for protocol description *SME* is given in Figure 11. The mediator is at the same time a requestor (since it requests an interaction at the query message) and an acceptor (since it accepts the operator's request when sending the calc message). We just

$$
\begin{array}{l}
\mathit{Inv}_{SME}(s) := \textsf{\textbf{acc}}(\textsf{\textbf{this}}.\textsf{mstate}) * \textsf{\textbf{acc}}(\textsf{\textbf{this}}.\textsf{c}) * \textsf{\textbf{acc}}(\textsf{\textbf{this}}.\textsf{op}) * \\
s = \mathbf{QM} \Rightarrow \textsf{\textbf{this}}.\textsf{mstate} = \textsf{ACCEPTQUERY} * \mathit{SME}^{F}(\textsf{\textbf{this}}, 0^{-}) * \\
s \in \{\mathbf{DM}, \mathbf{GM}, \mathbf{RM}\} \Rightarrow \\
\left(\begin{array}{l}
\textsf{\textbf{this}}.\textsf{c} = \varphi_{SME}(\textsf{\textbf{this}}, \mathbf{QM}, \iota_{SME}(\textsf{\textbf{this}}), \textsf{query(c,n)}, \textsf{c}) * \\
\left(\begin{array}{l}
\iota_{SC}(\textsf{\textbf{this}}.\textsf{c}) = \\
\varphi_{SME}(\textsf{\textbf{this}}, \mathbf{QM}, \iota_{SME}(\textsf{\textbf{this}}), \textsf{query(c,n)}, \iota_{SC}(\textsf{c}))
\end{array} \right)
\end{array} \right) \\
s \in \{\mathbf{DM}, \mathbf{GM}\} \Rightarrow \\
\left(\begin{array}{l}
\textsf{\textbf{this}} = \varphi_{SO}(\textsf{\textbf{this}}.\textsf{op}, \mathbf{CO}, \iota_{SO}(\textsf{\textbf{this}}.\textsf{op}), \textsf{calc(m,n)}, \textsf{m}) * \\
\left(\begin{array}{l}
\iota_{SME}(\textsf{\textbf{this}}) = \\
\varphi_{SO}(\textsf{\textbf{this}}.\textsf{op}, \mathbf{CO}, \iota_{SO}(\textsf{\textbf{this}}.\textsf{op}), \textsf{calc(m,n)}, \iota_{SME}(\textsf{m}))
\end{array} \right) * \\
\left(\begin{array}{l}
\varphi_{SC}(\textsf{\textbf{this}}.\textsf{c}, \mathbf{CO}, \iota_{SC}(\textsf{\textbf{this}}.\textsf{c}), \textsf{query(c,n)}, \textsf{n}) = \\
\varphi_{SO}(\textsf{\textbf{this}}.\textsf{op}, \mathbf{CO}, \iota_{SO}(\textsf{\textbf{this}}.\textsf{op}), \textsf{calc(m,n)}, \textsf{n})
\end{array} \right)
\end{array} \right) \\
s = \mathbf{DM} \Rightarrow \\
\left(\begin{array}{l}
\textsf{\textbf{this}}.\textsf{mstate} = \textsf{WAITFORDONE} * \mathit{SME}^{F}(\textsf{\textbf{this}}, 2^{-}) * \\
\left(\begin{array}{l}
\textsc{send}_{SC}(\textsf{\textbf{this}}.\textsf{c}, \iota_{SC}(\textsf{\textbf{this}}.\textsf{c}), \mathbf{AC}, \textsf{advance}) \;. \\
\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{GM}, \textsf{getresult}) \;. \\
\textsc{send}_{SC}(\textsf{\textbf{this}}.\textsf{c}, \iota_{SC}(\textsf{\textbf{this}}.\textsf{c}), \mathbf{RC}, \textsf{cresult}) \;. \textsc{EndR}
\end{array} \right) * \\
\left(\begin{array}{l}
\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{DM}, \textsf{done}) \;. \\
\textsc{send}_{SO}(\textsf{\textbf{this}}.\textsf{op}, \iota_{SO}(\textsf{\textbf{this}}.\textsf{op}), \mathbf{GO}, \textsf{get}) \;. \\
\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{RM}, \textsf{mresult}) \;. \textsc{EndS}
\end{array} \right)
\end{array} \right) \\
s = \mathbf{GM} \Rightarrow \\
\left(\begin{array}{l}
\textsf{\textbf{this}}.\textsf{mstate} = \textsf{ACCEPTGETRESULT} * \mathit{SME}^{F}(\textsf{\textbf{this}}, 2^{-}) * \\
\left(\begin{array}{l}
\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{GM}, \textsf{getresult}) \;. \\
\textsc{send}_{SC}(\textsf{\textbf{this}}.\textsf{c}, \iota_{SC}(\textsf{\textbf{this}}.\textsf{c}), \mathbf{RC}, \textsf{cresult}) \;. \textsc{EndR}
\end{array} \right) * \\
\left(\begin{array}{l}
\textsc{send}_{SO}(\textsf{\textbf{this}}.\textsf{op}, \iota_{SO}(\textsf{\textbf{this}}.\textsf{op}), \mathbf{GO}, \textsf{get}) \;. \\
\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{RM}, \textsf{mresult}) \;. \textsc{EndS}
\end{array} \right)
\end{array} \right) \\
s = \mathbf{RM} \Rightarrow \\
\left(\begin{array}{l}
\textsf{\textbf{this}}.\textsf{mstate} = \textsf{WAITFORMRESULT} * \mathit{SME}^{F}(\textsf{\textbf{this}}, 1^{-}) * \\
\textsc{send}_{SC}(\textsf{\textbf{this}}.\textsf{c}, \iota_{SC}(\textsf{\textbf{this}}.\textsf{c}), \mathbf{GO}, \textsf{cresult}) * \\
\textsc{rcv}_{SME}(\textsf{\textbf{this}}, \iota_{SME}(\textsf{\textbf{this}}), \mathbf{RM}, \textsf{mresult}) \;. \textsc{EndS}
\end{array} \right)
\end{array}
$$

Figure 11: Protocol invariant for SME, used for the mediator given in Figure 8.

briefly describe the mediator's protocol invariant without arguing about validity. The validity of the message handlers can be ensured similarly as shown for the operator (the requestor side) and as we are going to show for the customer (the acceptor side). The only difference in terms of proving validity is that the session predicate required to send the calc message to the operator is obtained right after the service has been spawned over the constructor postcondition.

The protocol invariant together with the preconditions ensures that no failing branches are taken. In the **GM** state the protocol invariant holds two interaction permissions. One is obtained via the request at the query message (we call this the *customer interaction*) and the other is obtained via the accept at the calc message (we call this the *service interaction*). None of the interactions have been advanced yet in this state.

After receiving the done message the mediator can advance the customer interaction as well as the operator interaction. This is reflected in the protocol invariant for session state **DM**. Next, after receiving the getresult message, the mediator can again advance both interactions. At this point it gets back the finalization permisson from the customer interaction (hence only one access permission is still missing). This is reflected in the protocol invariant for session state **RM**. Finally, once the mresult message is received the operator interaction is ended and the remaining finalization permission is retrieved. At this point the mediator can finish the session.

Some important facts are expressed using environment expressions. One important fact is that the value that was sent to the operator over the calc message in the operator's current session is the same value as the mediator received over the init message of its current session. The reason this is important is that using this fact the mediator can relate the value that it receives from the operator (which is expressed using the operator's environment) in terms of its own environment. The client which knows how its own environment relates to mediator's environment can then use this fact to express the value it gets in terms of its own environment. Without the mediator as a bridge, this chaining of the operator's and the client's environment would not be possible.

This mediator's protocol invariant is self-framing, partly because whenever one is in a state where a session identifier attribute function is used for a session other than the mediator's, then in that state the protocol invariant contains an interaction permission in which the corresponding session is active.

4.10.3 Customer specification

The precondition of the init message handler is given by

$$\begin{aligned} SC(\textbf{this}) * state_{SC}(\textbf{this}) = \mathbf{IC} * \\ SME(\mathsf{mediator}) * state_{SC}(\mathsf{mediator}) = \mathbf{QM} \end{aligned}$$

The preconditions of the $\mathsf{advance}$ and $\mathsf{cresult}$ message handlers only contain the customer's session predicate $SC(\textbf{this})$ in the session states $\mathbf{IC}$ (for init), $\mathbf{AC}$ (for $\mathsf{advance}$) and $\mathbf{RC}$ (for $\mathsf{cresult}$). We define the following protocol invariant for protocol description SC.

$$\begin{aligned}
Inv_{SC}(s) := {}& \textbf{acc}(\textbf{this}.\mathsf{cstate}) * \textbf{acc}(\textbf{this}.\mathsf{md}) * \\
& s = \mathbf{IC} \Rightarrow \textbf{this}.\mathsf{cstate} = \mathsf{INIT} * SO^F(\textbf{this}, 0^-) * \\
& s = \mathbf{AC} \Rightarrow \\
& \left(\begin{array}{l}
\textbf{this}.\mathsf{cstate} = \mathsf{WAITFORADVANCE} * SO^F(\textbf{this}, 1^-) * \\
\textbf{this} = \varphi_{SME}(\textbf{this}.\mathsf{md}, \mathbf{CO}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathsf{query(c,n)}, \mathsf{c}) * \\
\left(\begin{array}{l} \iota_{SC}(\textbf{this}) = \\ \varphi_{SME}(\textbf{this}.\mathsf{md}, \mathbf{CO}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c})) \end{array} \right) * \\
\left(\begin{array}{l} \varphi_{SC}(\textbf{this}, \mathbf{IC}, \iota_{SC}(\textbf{this}), \mathsf{init(n)}, \mathsf{n}) = \\ \varphi_{SME}(\textbf{this}.\mathsf{md}, \mathbf{QM}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathsf{query(c,n)}, \mathsf{n}) \end{array} \right) \\
\left(\begin{array}{l} \textsc{rcv}_{SC}(\textbf{this}, \iota_{SC}(\textbf{this}), \mathbf{AC}, \mathsf{advance}) \,. \\ \textsc{send}_{SME}(\textbf{this}.\mathsf{md}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathbf{GM}, \mathsf{getresult}) \,. \\ \textsc{rcv}_{SC}(\textbf{this}, \iota_{SC}(\textbf{this}), \mathbf{RC}, \mathsf{cresult}) \,. \textsc{EndS} \end{array} \right)
\end{array} \right) \\
& s = \mathbf{RC} \Rightarrow \\
& \left(\begin{array}{l}
\textbf{this}.\mathsf{cstate} = \mathsf{WAITFORCRESULT} * SO^F(\textbf{this}, 1^-) * \\
\textsc{rcv}_{SC}(\textbf{this}, \iota_{SC}(\textbf{this}), \mathbf{RC}, \mathsf{cresult}) \,. \textsc{EndS}
\end{array} \right)
\end{aligned}$$

The idea of the customer's protocol invariant is as follows. When the customer moves to session state $\mathbf{AC}$ (after sending the query message to the mediator) the interaction with the mediator has started. The facts in the protocol invariant for that state relate the customer's local state with the mediator's session environment (at the end of this section, we show how these facts can be established). One important fact is that the value it received over the init message is the same as the value sent to the mediator of the query message. This will be important to derive the response property modularly, because it permits a derivation of the mediator's reaction to the customer's messages just in terms of the mediator's environment.

Also note that in this session state the interaction permission obtained after sending the query message has not progressed, because the customer is still waiting to receive the $\mathsf{advance}$ message.

Consider message handler init. When sending the query message to the mediator, the customer needs to accept the requested interaction specified by the interaction permission (4.10). For this it needs to give up its own session finalization access permission, after which its left with $SO^F(\textbf{this}, 1^-)$. Furthermore, the customer must give up the send permission corresponding to the first send event in the requested interaction. This the customer can achieve by progressing its own session to state $\mathbf{AC}$ and transforming the corresponding session predicate to the send permission.

Additionally, the customer must satisfy the precondition of query which it can do by giving up the mediator's session predicate that it received over the precondition of init.

Once the customer has sent the query message, the customer gets hold of the dual interaction permission of (4.10). Using this and $SO^F(\textbf{this}, 1^-)$ it can partially establish the protocol invariant in state $\mathbf{AC}$.

To justify the parts involving the environment expressions, we present a rule that we have not shown before. We explain this rule by example. The customer sends sends the query message to the mediator, while the mediator's session is in state $\mathbf{QM}$. This means the customer is actively generating the session event specified by $(SME, \textbf{this}.\mathsf{md}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathbf{QM}, \mathsf{query})$. Therefore, the customer knows the values that were sent to the mediator over this event, which means that the customer has knowledge about the session environment with respect to the query message (in a sense the customer is the mediator's environment in this case). Using the rule the customer can learn the following facts (which are tracked in the customer's protocol invariant):

$$\textbf{this} = \varphi_{SME}(\textbf{this}.\mathsf{md}, \mathbf{CO}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathsf{query(c,n)}, \mathsf{c}) * \left(\begin{array}{l} \iota_{SC}(\textbf{this}) = \\ \varphi_{SME}(\textbf{this}.\mathsf{md}, \mathbf{CO}, \iota_{SME}(\textbf{this}.\mathsf{md}), \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c})) \end{array} \right)$$

These facts make explicit that the customer sent its own identity via the query message to the mediator and that the customer's session identifier is the same as when it sent the query message. Using these facts the customer can relate itself to the mediator's session environment, which will be important once we derive actor services.

Formally, this rule can be explained as follows. Suppose an actor a sends a message m associated with a protocol ρ to another actor a' using the statement $a'.m(\overrightarrow{e_i})$. By sending this message a gains knowledge about the session environment in a'. We make this explicit using the following rule. a must provide expressions e_{id}, e_s for the session identifier and session state for the session corresponding to ρ in a'. It must be guaranteed that e_{id}, e_s are framed after the message is sent (i.e. after all the permissions specified in the precondition have been given up). At the program point after sending the message the following assertion holds

$$\varphi_\rho(a', e_{id}, e_s, m(y; \overrightarrow{x_i}), e) = e[\overrightarrow{e_i}/\overrightarrow{x_i}][a'/this]$$

if the assertion is framed at that program point (i.e. the environment expression must be framed and $e[\overrightarrow{e_i}/\overrightarrow{x_i}][a'/y]$ must be framed). e can be picked in the rule (as long the framing conditions hold), and arbitrary many such assertions can be learnt when sending a single message.

The presented rule is also required to verify the validity of the query message handler in the mediator (the mediator gains knowledge about the operator's session environment when sending the calc message).

The customer's protocol invariant is self-framing, partly because in state **AC** an interaction permission is held which frames $\iota_{SME}(\textbf{this}.\mathsf{md})$. The preconditions together with protocol invariant ensure that no failing branches are taken in the message handlers.

4.10.4 Proving the response property

Using the specifications from the previous sections, we now show how to derive the actor service (4.9). We have the following local actor service for the mediator

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n_q}) \rightsquigarrow \exists O'_p.\ O'_p.\mathsf{calc}(M, n_q)\ \textit{where}\ \textbf{old}(\iota_{SME}(M)) = \iota_{SME}(M) \tag{4.11}$$

$\iota_{SME}(M)$ is framed, because the request clause requires the mediator's send event permission from the acceptor, hence one can think of this permission as part of the precondition of compute. We have the following local actor service for the operator

$$\underline{O_p}.\mathsf{calc}(\underline{M}, \underline{n_c}) \rightsquigarrow M.\mathsf{done}(f(n_c))\ \textit{where}\ \textbf{old}(\iota_{SME}(M)) = \iota_{SME}(M) \tag{4.12}$$

We compose the mediator's local actor service with an instantiation of the operator's local actor service to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n_q}) \rightsquigarrow M.\mathsf{done}(f(n_q))\ \textit{where}\ \textbf{old}(\iota_{SME}(M)) = \iota_{SME}(M) \tag{4.13}$$

Note how both of these actor services make explicit that the mediator's session does not change between trigger and response. We rewrite (4.13) to (using the rules introduced in Section 4.5)

$$\begin{aligned} &\underline{M}.\mathsf{query}(\underline{C}, \underline{n_q}) \rightsquigarrow M.\mathsf{done}(f(n_q)) \\ &\qquad\qquad \textbf{old}(\iota_{SME}(M)) = \iota_{SME}(M) \\ &\textit{where}\quad \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{c}) = C \\ &\qquad\qquad \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c})) = \textbf{old}(\iota_{SC}(C)) \end{aligned} \tag{4.14}$$

The main idea of this rewrite step is to relate the mediator's session environment to the inputs that it received over the query message (which corresponds to a session event for the mediator's session). This is important, because in this actor service we know the values that were provided for the query message. The next actor service that we derive, will express its response only in terms of the mediator's session environment. Hence we need to make sure that we can make the connection between the actual values and the specified environment in that response.

Note that $\textbf{old}(\iota_{SC}(C))$ is framed for a similar reason to why $\iota_{SME}(M)$ is framed in (4.11). Furthermore, we can derive the following local actor service for the mediator

$$\begin{array}{l} \underline{M}.\mathsf{done}(\underline{n_d}) \rightsquigarrow \exists C'.\ C'.\mathsf{advance}(n_d) \\ \textit{where}\ \begin{array}{l} \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{c}) = C' * \\ \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c})) = \iota_{SC}(C') \end{array} \end{array} \tag{4.15}$$

We compose (4.14) with an instantiation of (4.15) to get

$$\underline{M}.\mathsf{query}(\underline{C}, \underline{n_q}) \rightsquigarrow C.\mathsf{advance}(f(n_q))\ \textit{where}\ \textbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \tag{4.16}$$

We can derive the following local actor service for the customer

$$\underline{C}.\mathsf{init}(\underline{n_i}) \rightsquigarrow \exists M'.\ M'.\mathsf{query}(C, n_i)\ \textit{where}\ \textbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \tag{4.17}$$

We compose (4.17) with (4.16) to get

$$\underline{C}.\mathsf{init}(\underline{n_i}) \rightsquigarrow C.\mathsf{advance}(f(n_i))\ \textit{where}\ \textbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \tag{4.18}$$

This actor service shows that if the customer receives an init message, then eventually it will receive an advance message in the same session, i.e. its protocol is guaranteed to continue. The interesting part is that the actor service contains no information about the mediator and the operator. The operator's val field evaluates to the n_i argument given in the init message handler while the session is running. We will now see in the rest of the derivation how this fact is recovered without keeping track of it in (4.18), which makes the derivation modular. We have the following local service for the mediator

$$\begin{array}{l} \underline{M}.\mathsf{getresult}(\underline{n_{gr}}) \rightsquigarrow \exists O'_p.\ O'_p.\mathsf{get}(n_{gr}) \\ \textit{where}\ \begin{array}{l} M = \varphi_{SO}(M, \iota_{SO}(O'_p), \mathbf{QM}, \mathsf{calc(m,n)}, \mathsf{m}) * \\ \textbf{old}(\iota_{SME}(M)) = \varphi_{SO}(M, \iota_{SO}(O'_p), \mathbf{QM}, \mathsf{calc(m,n)}, \iota_{SME}(\mathsf{m})) * \\ \left(\begin{array}{l} \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{n}) = \\ \varphi_{SO}(M, \iota_{SO}(O'_p), \mathbf{QM}, \mathsf{calc(m,n)}, \mathsf{n}) \end{array} \right) \end{array} \end{array} \tag{4.19}$$

This local actor service makes explicit that the n_q value which the mediator received over its query message in its session was sent to the operator over the calc message. Next, we have the following local actor service for the operator

$$\begin{array}{l}\underline{O_p}.\mathsf{get}(\underline{n_{get}}) \rightsquigarrow \exists M', z.\ M'.\mathsf{mresult}(h(n_{get}+z)) \\ \quad \textit{where} \begin{array}{l}\varphi_{SO}(O_p, \textbf{old}(\iota_{SO}(O_p)), \mathbf{QM}, \mathsf{calc(m,n)}, \mathsf{m}) = M' * \\ \varphi_{SO}(O_p, \textbf{old}(\iota_{SO}(O_p)), \mathbf{QM}, \mathsf{calc(m,n)}, \iota_{SME}(\mathsf{m})) = \iota_{SME}(M') * \\ \varphi_{SO}(O_p, \textbf{old}(\iota_{SO}(O_p)), \mathbf{QM}, \mathsf{calc(m,n)}, \mathsf{n}) = z\end{array}\end{array} \tag{4.20}$$

The existentially quantified z is the value of the operator's val field which in this actor service is expressed using the operator's environment. We compose (4.19) with (4.20) to get

$$\begin{array}{l}\underline{M}.\mathsf{getresult}(\underline{n_{gr}}) \rightsquigarrow \exists z.\ M.\mathsf{mresult}(h(n_{gr}+z)) \\ \quad \textit{where} \begin{array}{l}\varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{n}) = z \\ \textbf{old}(\iota_{SME}(M)) = \iota_{SME}(M)\end{array}\end{array} \tag{4.21}$$

In this actor service, the existentially quantified variable z still has the same value as before, but we express it using the mediator's environment because we cannot express it anymore in terms of the operator's environment since the operator is not present in the trigger and also not in the response. We have now essentially made the connection between the operator's val field and the mediator's environment (without ever mentioning the val field). Next, we have the following local actor service for the mediator

$$\begin{array}{l}\underline{M}.\mathsf{mresult}(\underline{n_{mr}}) \rightsquigarrow \exists C'.\ C'.\mathsf{cresult}(n_{mr}) \\ \quad \textit{where} \begin{array}{l}\varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{c}) = C' \\ \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c})) = \iota_{SC}(C')\end{array}\end{array} \tag{4.22}$$

We compose (4.21) with an instantiation of (4.22) to get

$$\begin{array}{l}\underline{M}.\mathsf{getresult}(\underline{n_{gr}}) \rightsquigarrow \exists C', z.\ C'.\mathsf{cresult}(h(n_{gr}+z)) \\ \quad \textit{where} \begin{array}{l}\varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{n}) = z \\ \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{c}) = C' \\ \varphi_{SME}(M, \textbf{old}(\iota_{SME}(M)), \mathbf{QM}, \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c})) = \iota_{SC}(C')\end{array}\end{array} \tag{4.23}$$

We have the following local actor service for the customer

$$\begin{array}{l}\underline{C}.\mathsf{advance}(\underline{n_a}) \rightsquigarrow \exists M'.\ M'.\mathsf{getresult}(g(n_a)) \\ \quad \textit{where} \begin{array}{l}\left(\begin{array}{l}\varphi_{SC}(C, \textbf{old}(\iota_{SC}(C)), \mathbf{IC}, \mathsf{init(n)}, \mathsf{n}) = \\ \varphi_{SME}(M', \iota_{SME}(M'), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{n})\end{array}\right) * \\ C = \varphi_{SME}(M', \iota_{SME}(M'), \mathbf{QM}, \mathsf{query(c,n)}, \mathsf{c}) * \\ \textbf{old}(\iota_{SC}(C)) = \varphi_{SME}(M', \iota_{SME}(M'), \mathbf{QM}, \mathsf{query(c,n)}, \iota_{SC}(\mathsf{c}))\end{array}\end{array} \tag{4.24}$$

We compose (4.24) with an instantiation of (4.23) to get

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{C}.\mathsf{advance}(\underline{n_a}) \rightsquigarrow \exists z.\ C.\mathsf{cresult}(h(g(n_a)+z))} \\ \textit{where} & \begin{array}{l} \mathbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \\ \varphi_{SC}(C, \mathbf{old}(\iota_{SC}(C)), \mathbf{IC}, \mathsf{init}(\mathsf{n}), \mathsf{n}) = z \end{array} \end{array} \tag{4.25}$$

In this actor service we have now expressed the existentially quantified variable z in terms of the client's environment. In particular, we have now recovered the fact that the value stored in the operator's val, which was used during the operator's session to keep track of the initial value received over the calc message, evaluated to the argument that the client received over the init message. This actor service describes the second part of the client's protocol, ensuring that if the client reaches the beginning of this part, then the protocol is guaranteed to continue. We can now rewrite (4.18) (using the rule introduced in Section 4.5), which describes the first part of the protocol, to

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{C}.\mathsf{init}(\underline{n_i}) \rightsquigarrow C.\mathsf{advance}(f(n_i))} \\ \textit{where} & \begin{array}{l} \mathbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \\ \varphi_{SC}(C, \mathbf{old}(\iota_{SC}(C)), \mathbf{IC}, \mathsf{init}(\mathsf{n}), \mathsf{n}) = n_i \end{array} \end{array} \tag{4.26}$$

We compose (4.26) with an instantiation of actor service (4.25) describing the second part of the protocol to get

$$\underline{C}.\mathsf{init}(n_i) \rightsquigarrow C.\mathsf{cresult}(h(g(f(n_i)) + n_i))\ \textit{where}\ \mathbf{old}(\iota_{SC}(C)) = \iota_{SC}(C)$$

which is the actor service we wanted to derive.

4.11 Towards more general actor topologies

In this section we analyze an actor program that has a fundamentally different topology compared to the actor programs we have considered up to this point. As we will see, our approach cannot deal with this program. We point out how one can potentially generalize the current technique to deal with the program.

Consider the actor program represented by the message sequence chart in Figure 12. Each of the actors changes its behaviour according to a protocol. We assume that each actor only accepts the next message that it expects in the protocol, otherwise there is a failure. At the beginning the manager knows the identities of the server and the client. The manager sends the identities over messages to the client and the server, which reply that they are ready (this is done sequentially). Once the manager knows that the client

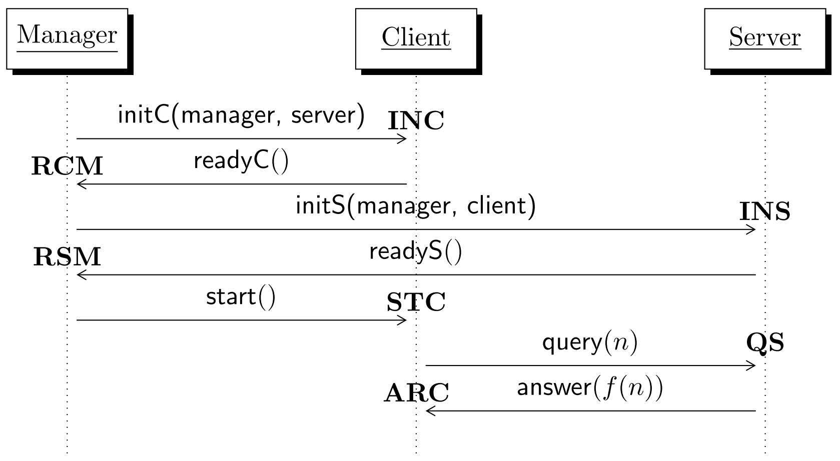

Figure 12: Message sequence chart showing an interaction between a client and a server that is setup by a manager. For each message the session state in which it is accepted by the receiving actor is annotated.

and the server are ready, the manager sends a message to the client indicating that the interaction with the server can be started.

There is a fundamental difference to the programs we have been considering so far. The client cannot setup the interaction with the server directly, because there is no initial message between the client and the server, where identities are exchanged. This setup is done via the manager.

Next, we focus on the problem of making sure that each of the actors gets the required send event permissions at the right points in their protocol. In Section 4.11.3 we discuss an issue related to deriving actor services in this setting.

Let us analyze how we would attempt to write specifications for the actors using our current technique. We define the protocol descriptions *SMA* for the manager, *SCL* for the client and *SRV* for the server. The transition relations are given by

$$\mathbf{RCM} \sqsubset_{SMA} \mathbf{RSM}$$

$$\mathbf{INC} \sqsubset_{SCL} \mathbf{STC} \sqsubset_{SCL} \mathbf{ARC}$$

$$\mathbf{INS} \sqsubset_{SRV} \mathbf{QS}$$

The association of session states to message handlers is straightforward. For example, the server accepts message initS in state **INS** and message query in state **QS**. We assume the preconditions of the message handlers are defined accordingly. For example, the precondition of the initS message handler is given by

$$\mathit{SRV}(\textbf{this}) * \mathit{state}_{\mathit{SRV}}(\textbf{this}) = \mathbf{INITS}$$

We need to make sure that each of the actors has the necessary permission to send the respective messages. We assume the manager at the beginning has the session predicates required to send the initC and initS messages. Once the client receives the start message, it requires the session predicate to send the query message to the server. It is not easily possible, without hiding the implementation details of the client, to specify such a session predicate in the precondition of the start message handler. The main reason is that there is no information on the server's identity in this message (this is analogous to the discussion in Section 4.7). Our solution was to introduce request clauses, with which one can request interactions at the points where one gets to know the identities of the other communicating parties. For the client, this point is when it receives the initC message. We could provide the following request clause for the initC message specifying the client's interaction:

$$\begin{aligned}
&\textsc{send}_{\mathit{SMA}}(\mathsf{manager}, \iota_{\mathit{SMA}}(\mathsf{manager}), \mathbf{RCM}, \mathsf{readyC})\ .\\
&\textsc{rcv}_{\mathit{SCL}}(\textbf{this}, \iota_{\mathit{SCL}}(\textbf{this}), \mathbf{STC}, \mathsf{start})\ .\\
&\textsc{send}_{\mathit{SRV}}(\mathsf{server}, \iota_{\mathit{SRV}}(\mathsf{server}), \mathbf{QS}, \mathsf{query})\ .\\
&\textsc{rcv}_{\mathit{SCL}}(\textbf{this}, \iota_{\mathit{SCL}}(\textbf{this}), \mathbf{ARC}, \mathsf{answer})\ .\\
&\textsc{EndS}
\end{aligned}$$

This request clause is different from the request clauses that we used for the CMO program (see Section 4.10), because here the specified interaction involves more than two actors. To accept the request when sending the initC message, the manager must give up session finalization access permission for the server's session. We assume that the manager has this permission and we point out in Section 4.11.2 how one could avoid this requirement. Hence the manager can accept this request, since the manager also can give up send permission for the readyC message (we assume the manager is in session state **RCM** when sending the message) and a session finalization access permission of its own session.

Analogously, we could provide the following request clause for initS, specifying the server's interaction message (which again involves all three parties):

$$\begin{aligned}&\textsc{send}_{SMA}(\mathsf{manager}, \iota_{SMA}(\mathsf{manager}), \mathbf{RSM}, \mathsf{readyS})\ .\\ &\textsc{rcv}_{SRV}(\textbf{\textsf{this}}, \iota_{SRV}(\textbf{\textsf{this}}), \mathbf{QS}, \mathsf{query})\ .\\ &\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{ARC}, \mathsf{answer})\ .\\ &\textsc{EndS}\end{aligned}$$

Upon receiving the initC message, the manager progresses its session to state **RSM**. Hence the master can accept this request when sending the initS message by giving up the send permission for the readyS message and the necessary session finalization permissions (we assume it has such a permission for the client's session).

When the manager accepts the requests, it gets hold of the interaction permissions that are dual to the interaction permissions specified in the request clauses. We assume it keeps track of these interaction permissions and their progressions in the protocol invariant (as in Section 4.10 for the CMO program). Once it receives the readyC and the readyS messages, it can progress the interaction permissions. Hence before sending the start message to the client, the manager holds the following interaction permissions

$$\begin{aligned}&\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{STC}, \mathsf{start})\ .\\ &\textsc{rcv}_{SRV}(\mathsf{server}, \iota_{SRV}(\mathsf{server}), \mathbf{QS}, \mathsf{query})\ .\\ &\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{ARC}, \mathsf{answer})\ .\\ &\textsc{EndR}\end{aligned} \tag{4.27}$$

$$\begin{aligned}&\textsc{send}_{SRV}(\mathsf{server}, \iota_{SRV}(\mathsf{server}), \mathbf{QS}, \mathsf{query})\ .\\ &\textsc{rcv}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{ARC}, \mathsf{answer})\ .\\ &\textsc{EndS}\end{aligned} \tag{4.28}$$

To send the start message to the client, the manager requires the corresponding send permission. If the manager could progress interaction permission (4.27), then it would get hold of the send permission for start. However, to progress interaction permission (4.27), the manager requires the send permission for the server's query message. If the manager could progress interaction permission (4.28), then it would get hold of this permission required for the query message. However, to progress interaction permission (4.28) the manager requires the send permission for the client's answer message, which it has no chance of obtaining at this point, because the client is in a session state where it does not accept the answer message. We conclude that our current rules for progressing interaction permissions are not sufficient such that the manager gets the necessary permission to send the start message.

Let us analyze the issue in more detail. One may progress interaction permission (4.28) and as a result get hold of the permission for the server's query message, if one guarantees that when the server receives this query message, then the server will be able to get hold of the permission for the client's answer message using the companion permission of (4.28). The current rule for progressing interaction permission (4.28) guarantees this by requiring that one gives up the permission for the client's answer message before one can get hold of the send permission for query. The permission that is given up can be thought of as being implicitly transferred to the companion permission held by the server. However, there are other ways of guaranteeing this, which are not captured by the rule.

Suppose we could get hold of the permission for the server's query message and would give it up to progress interaction permission (4.27) to its continuation. The permission for query would be then implicitly transferred to the companion interaction permission I_C of interaction permission (4.27) held by the client (in this case I_C is just the dual of interaction permission (4.27) as specified by Property 3). Once the client were to receive the start message, it could progress I_C to I'_C. I'_C is given by

$$\begin{aligned}&\textsc{send}_{SRV}(\mathsf{server}, \iota_{SRV}(\mathsf{server}), \mathbf{QS}, \mathsf{query})\ .\\&\textsc{rcv}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{ARC}, \mathsf{answer})\ .\\&\textsc{EndR}\end{aligned} \tag{I'_C}$$

To get hold of the permission for the server's query message by progressing this interaction, the client must give up the send permission for its answer message, which is then implicitly transferred to the continuation of interaction permission (4.27).

This means the interaction permission I_C gives us a guarantee, that if we were to give it the send permission for query, then this permission would not be used until the client would give up its send permission for answer. If we can now make sure that this send permission for answer, when given up, is implicitly transferred to the interaction permission held by the server instead of the continuation of (4.27) (which is held by the manager), then it would be fine to use interaction permission (4.28) to progress interaction permission (4.27).

This seems to suggest that it should be possible to repartition interaction permission (4.27) and interaction permission (4.28) to the following single interaction permission

$$\begin{aligned}&\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{STC}, \mathsf{start})\ .\\&\textsc{EndR}\end{aligned} \tag{4.29}$$

The send event for the answer message in interaction permission (4.27) has been removed. The idea is that by removing it from interaction permission (4.27), we make sure that when the client gives up its permission for the answer message to progress its interaction permission, then there is no way of obtaining the send permission for answer by progressing interaction permission (4.29). Instead, we in a sense are rewiring this permission to the server's interaction permission which is the companion permission of interaction permission (4.28). Finally, the manager can use the interaction permission (4.29) to get hold of the send permission to send the start message.

We only gave an here intuition as to why the presented repartitioning step should work. The justification given in Section 4.9 for the original repartitioning steps cannot be easily generalized for this scenario. The reason is that in this scenario, one cannot identify for each interaction permission a unique companion interaction permission to which it transfers permissions. For example, when the client makes the request, its companion permission is given by interaction permission (4.27) and after the final repartitioning step applied by the manager, its companion permission in a sense switches to the interaction permission held by the server. In Section 4.11.1 we consider a modified version of the program and we show that a similar repartitioning step such as the one presented here can be potentially used to obtain the required permissions.

4.11.1 A slightly modified version

In this section we consider a slightly modified version of the program represented by the message sequence diagram in Figure 12. Consider the program represented by the message sequence diagram in Figure 13. The only difference to Figure 12 is that at the end of the client's interaction with the server, the client sends a complete message to the manager. We assume the manager's protocol *SMA* has an additional state **CM** in which it accepts the complete message. The presented request clause for the client can be updated to (it only has a single event more)

$$\begin{aligned}
&\textsc{send}_{SMA}(\mathsf{manager}, \iota_{SMA}(\mathsf{manager}), \mathbf{RCM}, \mathsf{readyC})\ .\\
&\textsc{rcv}_{SCL}(\textbf{this}, \iota_{SCL}(\textbf{this}), \mathbf{STC}, \mathsf{start})\ .\\
&\textsc{send}_{SRV}(\mathsf{server}, \iota_{SRV}(\mathsf{server}), \mathbf{QS}, \mathsf{query})\ .\\
&\textsc{rcv}_{SCL}(\textbf{this}, \iota_{SCL}(\textbf{this}), \mathbf{ARC}, \mathsf{answer})\ .\\
&\textsc{send}_{SMA}(\mathsf{manager}, \iota_{SMA}(\mathsf{manager}), \mathbf{CM}, \mathsf{complete})\ .\\
&\textsc{EndR}
\end{aligned}$$

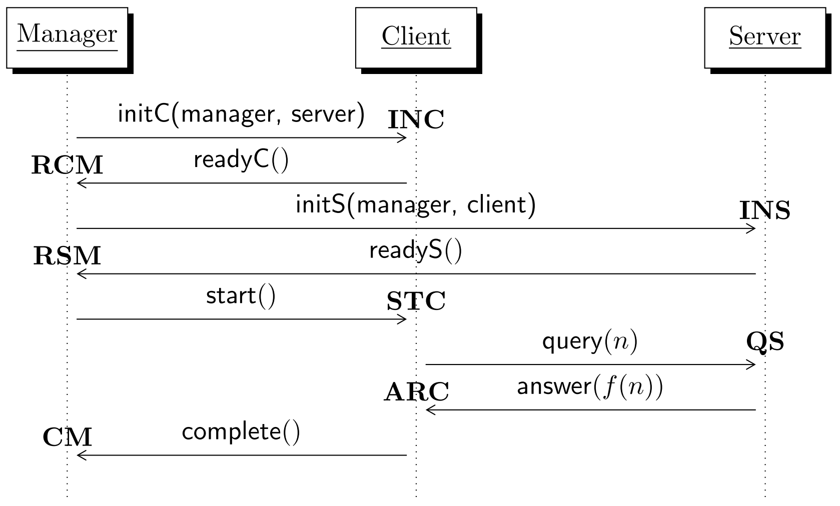

Figure 13: Modified version of Figure 12. In this version the client sends a message to the manager once its interaction with the server is finished. For each message the session state in which it is accepted by the receiving actor is annotated.

The manager can progress the received interaction permissions in the same way as before when receiving the readyC and readyS messages. At the point right before the manager sends the start message to the client, the manager holds interaction permission (4.30) and interaction permission (4.28).

$$\begin{aligned}&\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{STC}, \mathsf{start})\ .\\&\textsc{rcv}_{SRV}(\mathsf{server}, \iota_{SRV}(\mathsf{server}), \mathbf{QS}, \mathsf{query})\ .\\&\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{ARC}, \mathsf{answer})\ .\\&\textsc{rcv}_{SMA}(\mathsf{manager}, \iota_{SMA}(\mathsf{manager}), \mathbf{CM}, \mathsf{complete})\ .\\&\textsc{EndS}\end{aligned} \tag{4.30}$$

It needs to somehow extract the send permission for the start message from interaction permission (4.30). The main difference to the original version is that now there is an additional event at the end of this interaction unrelated to the server. Concretely, the difference is that we need to make sure that when the client eventually receives the answer message, then it can get hold of the send permission for the manager's complete message.

The idea is to repartition interaction permission (4.30) and interaction permission (4.28) to the single interaction permission

$$\begin{aligned}&\textsc{send}_{SCL}(\mathsf{client}, \iota_{SCL}(\mathsf{client}), \mathbf{STC}, \mathsf{start})\ .\\&\textsc{rcv}_{SMA}(\mathsf{manager}, \iota_{SMA}(\mathsf{manager}), \mathbf{CM}, \mathsf{complete})\ .\\&\textsc{EndS}\end{aligned} \tag{4.31}$$

The reasons why this repartition step should be permitted are the following. The first reason is analogous to how we argued in the original case. We may use the interaction permission (4.28) as send permission for query, because the interaction permission (4.30) guarantees that the obtained send permission will only be used once the send permission for answer has been given up (and we can rewire this permission to the server's interaction permission). The second reason is that since we keep the receive event for the manager's complete message in interaction permission (4.31), we cannot obtain the permission for the client's start message until this permission has been given up.

Before the manager sends the start message, the manager can progress its session to state **CM** and it can progress interaction permission (4.31) by giving up the send permission for the complete message, after which the manager gets hold of the permission to send the complete message.

What is interesting about this example, is that the client only requires the permission for complete once it receives the answer message from the server. However, with this repartitioning step, the manager can give up the corresponding permission at an earlier point. This permits information hiding, because it enables the client to write down its request clause from its point of view without having to know when the permission becomes ready. In our previous examples, the permission was always transferred at the point where it was expected and never before, because it was not possible (and also not required) to transfer it earlier.

This point becomes clear if one thinks about how one would handle the presented example without interaction permissions but by explicitly specifying the session predicates which need to be transferred in preconditions (ignoring the fact that one would probably have to expose implementation details). There are two logical options. The first option is that one specifies the session predicate for the complete message in the precondition of the client's start message handler, because that is where this permission is first ready. The manager could then send it over this message. This would however mean that the client knows when the manager reaches a state when this permission becomes ready.

The second option is that the client specifies the session predicate for the complete message in the precondition of the client's answer message handler, because at that point the client requires this session predicate. The disadvantage here is that the server would have to somehow get hold of this session predicate so that the answer message could be sent, which means the server needs to know about the complete message. Hence information is exposed to the server that it does not need to know about.

4.11.2 Introducing sessions at later points in an interaction

As we mentioned, for the manager to accept the client's interaction request it must give up a session finalization access permission for the server's session (because the server's session appears as a send event in the request clause). This is disadvantageous for two reasons. First, the manager needs to get hold of this permission from somewhere. Second, there is no easy way for the server to get back this session finalization access permission (because it never gets hold of the corresponding interaction permission).

The main issue is that we require that all the sessions in the request clause are active at the time of the request. This made sense in the CMO program where all the specified interactions were between two parties. However, once one has more than two parties and one of the parties only gets involved in the interaction at a later point (such as the server in the client's requested interaction), then this becomes a restriction, because it could very well be possible that one of the specified session is not yet active. One idea to get around this is to generalize interaction permissions with constructs that allow specifying from which point onwards the session should become active. For example, we could specify the request clause of the client as follows

$$\begin{aligned}
&\textsc{send}_{SMA}(\mathsf{manager}, \iota_{SMA}(\mathsf{manager}), \mathbf{RCM}, \mathsf{readyC})\ .\\
&\textsc{rcv}_{SCL}(\textbf{this}, \iota_{SCL}(\textbf{this}), \mathbf{STC}, \mathsf{start})\ .\\
&\textsc{intro}_{SRV}(\mathsf{server}, id_s)\ .\\
&\textsc{send}_{SRV}(\mathsf{server}, id_s, \mathbf{QS}, \mathsf{query})\ .\\
&\textsc{rcv}_{SCL}(\textbf{this}, \iota_{SCL}(\textbf{this}), \mathbf{ARC}, \mathsf{answer})\ .\\
&\textsc{EndS}
\end{aligned}$$

The construct $\textsc{intro}_{SRV}(\mathsf{server}, id_s)$ specifies that at this point in the interaction a session governed by SRV in server becomes active. id_s is a bound variable that can be used to refer to the session identifier that the session will have once the interaction reaches that point. Once the interaction permission is progressed to the point where the session becomes active one can instantiate id_s with $\iota_{SRV}(\mathsf{server})$.

With such a construct one can change the requirements for an actor accepting an interaction request to only have to give up session finalization permissions for the sessions specified in send events that are active in the request clause. However, justifying such an extension is not completely straightforward. One must guarantee that whenever a session is active in an interaction permission (where identifying active sessions in an interaction permission is different with the introduced constructs compared to the original approach), then the session is guaranteed to be running and will not finish as long as it is active in the interaction permission. We leave this extension and justification for future work.

4.11.3 Knowledge about the environment

For the program represented by the message sequence diagram given in Figure 12, we have only considered how one could make sure that the different participants get hold of the required permissions at the different points in their protocol. Next, we consider the main issue that the program poses with respect to actor services that one should be able to derive, assuming that one has a solution for the permissions. One would expect that one can derive the following actor service

$$\underline{C}.\mathsf{start}() \rightsquigarrow \exists y.\ C.\mathsf{answer}(y)\ \textit{where}\ \textbf{old}(\iota_{SCL}(C)) = \iota_{SCL}(C) \tag{4.32}$$

Let us try to derive it. We can derive the following local actor service for the client

$$\begin{aligned}&\underline{C}.\mathsf{start}() \rightsquigarrow \exists S, x.\ S.\mathsf{query}(x)\\ &\quad \textit{where}\ S = \varphi_{SCL}(C, \textbf{old}(\iota_{SCL}(C)), \mathbf{INC}, \mathsf{initC(m,srv)}, \mathsf{srv})\end{aligned} \tag{4.33}$$

and we can derive the following local actor service for the server

$$\begin{aligned}&\underline{S}.\mathsf{query}(\underline{x}) \rightsquigarrow \exists C'.\ C'.\mathsf{answer}(f(x))\\ &\quad \textit{where}\ C' = \varphi_{SO}(S, \textbf{old}(\iota_{SRV}(S)), \mathbf{INS}, \mathsf{initS(m,c)}, \mathsf{c})\end{aligned} \tag{4.34}$$

Composing the two gives us

$$\underline{C}.\mathsf{start}() \rightsquigarrow \exists C', y.\ C'.\mathsf{answer}(y) \tag{4.35}$$

Actor service (4.35) is different to the desired actor service (4.32), since we do not know which client gets the answer and we do not know if the client's session is still the same. The reason we cannot derive these properties is because the client does not know anything about the server it is talking to (which it initially received over the initC message). In particular, it does not know the following fact

$$C = \varphi_{SO}(S, \iota_{SRV}(S), \mathbf{INS}, \mathsf{initS(m,c)}, \mathsf{c})$$

where S is the server the client C is talking to. This fact expresses that the client which the server received over the initS message evaluates to C. The reason the client does not know this is because currently the only way this fact could have been learned by the client is if the client sent the initS message to the server. However, the manager was the one sending this message. In the CMO program which was introduced in Section 4.7, for example, the mediator knows which mediator the operator is talking to, since the mediator is the one that provided its own identity to the operator.

One direction that one could follow to deal with this issue is to make certain facts explicit in the interaction permissions. For example, one could state that once the client C receives the $\mathsf{start}()$ message, we have that the server is also talking to C. This means we would be making facts about the environments of the different participants explicit in the interaction permission.

4.12 A suitable class of actor programs

As we observed in Section 4.11 the developed technique does not quite work yet for all kinds of actor programs. In this section we identify a class of protocol-based actor programs for which the technique is effective.

In Section 4.10 we showed that the technique works well for the CMO actor program. The topology of the CMO actor program is very structured. The customer communicates with the mediator, the mediator communicates with the operator and there never is any communication between the customer and the operator. This setup permits the modular derivation of the response properties. To make this clearer, recall that in Section 4.10, we were able to derive the following actor service (RP1) describing the first response property only involving the customer

$$\underline{C}.\mathsf{init}(\underline{n_i}) \rightsquigarrow C.\mathsf{advance}(f(n_i)) \textit{ where } \mathbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \tag{RP1}$$

Furthermore, we were able to independently derive the following actor service describing the second response property (RP2) only involving the customer

$$\begin{array}{c}\underline{C}.\mathsf{advance}(\underline{n_a}) \rightsquigarrow \exists z.\ C.\mathsf{cresult}(h(g(n_a)+z)) \\ \textit{where} \begin{array}{l}\mathbf{old}(\iota_{SC}(C)) = \iota_{SC}(C) \\ \varphi_{SC}(C, \mathbf{old}(\iota_{SC}(C)), \mathbf{IC}, \mathsf{init}(\mathsf{n}), \mathsf{n}) = z\end{array}\end{array} \tag{RP2}$$

We could then compose these two actor services to get the main response property in terms of the argument n_i given in the init message.

As we noted in Section 4.10, between the init message in actor service (RP1) being received and the $\mathsf{advance}$ message in the same actor service being sent, there are multiple stateful changes in the mediator and the customer that are dependent on n_i. However, we do not need to track them at all in actor service (RP1) (which enables us to write down a modular actor service) and yet we are able to use the derived actor service for the final result.

The main reason this works is that the customer knows how its own session environment relates to the mediator and the mediator knows how its own session environment relates to the operator. By session environment we mean the input values that actors receive over messages during sessions. Furthermore, the customer C knows that the mediator is talking to the customer C because the customer provided this information for the mediator's session environment. A similar observation can be made for the mediator and the operator. For the overall response property we are interested in how the stateful changes in the mediator and the operator relate to the final value in terms of the initial value n_i in the customer's session environment.

When we derive the mediator's reaction to the advance message which eventually leads to another message to the customer, we can describe the reaction that involves the operator at some point in terms of the mediator's environment since the mediator knows its relation to the operator's environment. Finally, since the customer knows its relation to the mediator's environment one can then describe the complete reaction just in terms of the customer's environment. This can be seen in actor service (RP2) where the final value given by the existentially quantified variable z is expressed in terms of the customer's environment. So the stateful changes that occur in the mediator and service during actor service (RP1) are captured in terms of the different environments, which permits the modular derivation.

More generally, a subset of such hierarchical topologies, where one has the same features as described, are dynamic tree topologies, where an actor is a node in the tree. Each actor interacts directly with its parent (except if it is the root) and with its children. In the CMO program the customer is the root with the mediator as its only child and the mediator has the operator as a child node. New actors (i.e. nodes in the tree) may be spawned at any time. An actor knows how its session environment relates to its childrens' session environment. In the CMO program, we could add arbitrarily many other types of actors that the mediator talks to (these actors being children of the mediator) and one could still reason about the overall property modularly.

Another feature of such tree topologies is that one can describe each interaction between a node and one of its children separately. The two nodes can setup this interaction over some initial messages where identities are exchanged, in our technique we capture this using request clauses. This means the request clauses that we introduce for such examples only involve interactions describing two parties. However, our technique permits dependencies between the different interactions in a single actor. These dependencies are implicit in the interaction permissions expressing the interactions. For example, after the mediator sends the calc message to the operator, it cannot directly continue the interaction with the customer, because it cannot give the customer a guarantee that it will accept getResult messages. Our current rules for progressing interaction permissions are well-suited for such interactions between two parties.

Our technique exploits the described features in a way that permits restricting the reasoning about a single actor's implementation to properties relating the actor to its parent node and to its children nodes. One never needs to reason about any other node in the tree other than the parent or the children when considering a single actor. This enables the modular derivation of response properties in such tree topologies. In particular, if messages are sent from node n_0 to its children and then at some point it gets a message back, then we can relate these two points involving n_0 independent of how far the information travelled down the tree, without any of the actors in the subtrees of the children knowing anything about n_0. In fact, we could change all the nodes that are not the parent or a child of n_0 and all the proved properties about n_0's implementation would still hold. Furthermore, we could change node n_0 and any response property that does not involve n_0 in its derivation would still hold in general[11]. We conclude that for such tree topologies our technique is effective, as long as the interactions do not involve features that we currently do not support (such as branching or recursive behaviour).

In the original actor services logic these modularity features cannot be achieved (to the best of our knowledge) without enforcing immutability on the part of the states that one wants to relate between the different points. As we saw in Section 3.2 in the original logic, one needs to pass information over messages about the parts of the actor's state that one wants to track, even if none of this information is required by any of the other actors. In our example this means if n_0 sends a message to one of its children c_0, then n_0 would have to send along information about its internal state (assuming n_0 expects a response at some point). Suppose upon receiving this message c_0 sends a message to one of its children c_0', which computes some value and then sends the value back to c_0, which finally sends it to n_0. The information about n_0's internal state would have to be sent to c_0'. Hence c_0' needs to know about n_0 in some way, therefore removing n_0 impacts the specification of c_0' and therefore also response properties that involve c_0' in their derivation but not n_0.

[11] An exception is if the response property depends on information that n_0 sent, then the response property may not hold anymore if n_0 is changed. However, in such a case the response property is inherently dependent on n_0 independent of our technique. We are saying that if the response property in a subtree does not depend on any specific information that n_0 sent, then we achieve the described modularity features.

In the example considered in Section 4.11 the topology is not a tree topology, since each of the actors interacts with each of the other actors. As a result, an actor may not know certain parts of another actor's environment that relate to it. For example, as discussed in Section 4.11.3, the client C that communicates with server S does not know that the client specified in the server's environment is given by C. Furthermore, the request clauses specify interactions with more than two parties, since in the example it is not the case that each pair of actors sets up their interaction independently. As we observed, dealing with such interactions requires more flexible ways to progress an interaction permission.

5 Towards modular fork-join reasoning in actor programs

When performing a computation, it is sometimes possible to divide the work into smaller subtasks and to combine the partial results of the subtasks to get the result of the complete computation. If an actor has to perform such a computation, it can delegate the subtasks to other actors which can in parallel compute the work. Eventually, an actor gets the partial results over messages and combines them to get the final result. This pattern is often referred to as the *fork-join pattern* (the forking corresponds to the delegation of the subtasks, while the joining corresponds to the combination of the partial results). One popular programming model used in practice that follows a similar style of computation is MapReduce [7].

Consider the actors given in Figure 14. The expected program flow is as follows. Once the master receives a queryA message specifying a client C and value n, its goal is to send $f(n) + g(n)$ to C. The master divides the work into two subtasks, namely computing $f(n)$ and $g(n)$. It delegates these subtasks to two different workers in the queryA message handler (this is the fork). The workers send the corresponding partial results back to the master over the subres message, which sums them up (this is the join). The partial results sent by the workers may be received in any of the two possible orders.

The master uses the state field to keep track of its current behaviour. If it is not performing a join, it accepts queryA messages (the field evaluates to ACCEPTQUERYA) and once it has received a queryA message, it only accepts subres messages (the field evaluates to WAITFORSUBRES). So the master is changing its behaviour according to a protocol. This is similar to the manager given in Figure 4 on page 13. One reason for this protocol-based behaviour is if the master were in a state where it is performing the join and it accepted queryA messages, then the information that it uses to keep track of the join would be reset, leading to wrong behaviour.

To perform the join the master tracks various information in its local state:

- The c field stores the client received in the query, such that once the computation is done, the master knows whom to send the result to. This is similar to previous examples.
- The result field sums up the results received from the workers.
- The counter field stores the number of subres messages that have been received by the master. If this value reaches two, then the master

```
actor Master {
    Client c; int counter; int result; State state;

    Master() { this.state := ACCEPTQUERYA; }
    handler queryA(Client client, int n) {
        if(this.state == ACCEPTQUERYA) {
            this.c := client;
            this.counter := 0;
            this.result := 0;
            Worker w1 := spawn WorkerA();
            Worker w2 := spawn WorkerB();
            w1.computeA(this,n);
            w2.computeB(this,n);
            this.state := WAITFORSUBRES;
        } else { fail(); }
    }
    handler subres(int r) {
        if(this.state == WAITFORSUBRES) {
            this.counter := this.counter+1;
            this.result := this.result+r;
            if(this.counter == 2) {
                this.c.sol(this.result);
                this.state := ACCEPTQUERYA;
            }
        } else { fail(); }
    }
}
actor WorkerA {
    handler computeA(Manager m, int n) {
        m.subres(f(n));
    }
}
actor WorkerB {
    handler computeB(Manager m, int n) {
        m.subres(g(n));
    }
}
```

Figure 14: Master forks work to two different workers and joins the results by summing them up. We assume the client actor has a sol *message handler.*

concludes that all expected messages have been received and the result field contains the final result. Hence in this case the master sends the solution to the client (see line 22), at which point it goes back to a state in which it accepts queries.

The response property summarizing the expected program flow can be expressed using the following actor service

$$\underline{M}.\mathsf{queryA}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n) + g(n)) \tag{5.1}$$

However, to ensure that this response property holds (or stated differently, that the program flow is always as expected) one needs to ensure:

- The subres messages sent by the workers are accepted by the master, otherwise the master will not get the expected partial results.
- No other subres messages are accepted by the master, otherwise the master will potentially get wrong partial results.
- The state required to track the join cannot be modified by message handlers other than the subres message handler (including any other message handlers in the master), otherwise the master will potentially compute a wrong result or may not even send the response.

These properties are similar to the properties we had to ensure for the manager in Figure 4. To deal with the manager, we introduced the notion of a session in Section 4, which is a concrete interaction specified by an abstract protocol. Sessions also are useful for the master in this example. We could define the interaction starting from the master being able to receive queryA messages until the sol message is sent to the client as one session. The properties that we need to ensure just boil down to the following:

- The workers must have partial ownership of the master's session once they receive compute and until they send the subres messages, ensuring that the master cannot progress its session during this time frame.
- No other actor should have partial ownership of the master's session.
- The permissions associated with the fields tracking the join should be contained in the corresponding protocol invariant, ensuring that no other message handler can modify the fields.

However, there are some features that we require here that the sessions introduced in Section 4 do not support. We require a way to give *multiple* (instead of just one) actors simultaneously partial session ownership that ensures the session cannot progress, which would allow for the master to receive the subres messages in any order. Furthermore, we need to know when the join is finished, because once a message associated with the join is received, one cannot progress the session except if it is known that all the expected join messages have been received. This requires capturing the state in the actor that tracks the join.

Assume that we can extend the sessions appropriately to capture the required properties. Let us analyze how to derive actor service (5.1) given such a session. As we have seen in previous sections, actor services can be derived modularly by composing other actor services that describe smaller parts of the behaviour. We can split the behaviour described by actor service (5.1) into the following four basic parts:

1. Whenever the master receives an expected queryA message, then eventually there will be two response messages (computeA, computeB) to the workers containing the same value n as in queryA.

2. Whenever WorkerA receives computeA(M, n), then eventually master M will get a subres message with value $f(n)$.

3. Whenever WorkerB receives computeB(M, n), then eventually master M will get a subres message with value $g(n)$.

4. Whenever the master receives the two expected subres messages with values r_1 and r_2 in the same session, then eventually the client that was received in the session event corresponding to the query message of the same session will get a sol message containing $r_1 + r_2$.

The second and third behaviours can be expressed using the following actor services (both of which can be derived in the original actor services logic)

$$\underline{W_A}.\mathsf{computeA}(\underline{M}, \underline{n}) \rightsquigarrow M.\mathsf{subres}(f(n)) \tag{5.2}$$

$$\underline{W_B}.\mathsf{computeB}(\underline{M}, \underline{n}) \rightsquigarrow M.\mathsf{subres}(g(n)) \tag{5.3}$$

However, actor services cannot express the first behaviour (the fork), because it specifies *multiple response messages* and they also cannot express the fourth behaviour (the join), because it specifies *multiple trigger messages*. The goal of this section is to make all the necessary extensions to the sessions and the actor services to prove response properties such as those expressed by actor service (5.1).

In Section 5.1 we extend the definition of actor services to contain multiple triggers and multiple responses. Next, in Section 5.2 we extend the sessions introduced in Section 4 to support the joining of messages during sessions. Using this extension, we then present rules to derive local actor services with multiple triggers and multiple responses in Section 5.3. In Section 5.5 we introduce rules to compose actor services with multiple triggers and multiple responses.

5.1 Actor services with multiple triggers and multiple responses

We generalize an actor service to be of the form:

$$\forall \overrightharpoon{X_j}.\, T \leadsto^{\mathcal{S}} R$$

where T is a *trigger pattern*, R is a *response pattern* and $\mathcal{S}$ is a *session association*. Trigger patterns are finite sets of trigger messages. In examples we denote a trigger pattern by

$$T_1 \times T_2 \times \ldots \times T_k$$

A trigger message T_i is a term $e.m(\overrightharpoon{e_i})$, where m is a message and e, $\overrightharpoon{e_i}$ are one-state expressions (they do not mention **old**). Response patterns are finite sets of *complete responses*. In examples we denote a response pattern by

$$c_1 \mid c_2 \mid \ldots \mid c_n$$

A complete response c is a finite set of *response messages*. In examples we denote a *complete response* by

$$r_1 \times r_2 \times \ldots \times r_{n'}$$

A response message[12] has the form

$$E.m(\overrightharpoon{E}) \ \textit{where}\ A$$

where m ranges over messages, E ranges over one-state expressions and A ranges over two-state assertions.

A session association $\mathcal{S}$ is a tuple (ρ, a) where ρ is a protocol description and a is an actor. It associates the actor service with sessions governed by ρ in a. If there is only one trigger message, then there is no session association and otherwise there is a session association (its significance will become clear later).

[12] In [20] response messages can also be empty responses. We omit empty responses, since we do not require them here.

Before we define the meaning of such an actor service in general, let us consider the following concrete actor service with two trigger messages and a single response message (ignoring the session association for now), which we assume to hold in the current program state

$$a.m(\underline{x_1}) \times a.m(\underline{x_2}) \rightsquigarrow b.m'(x_1 + x_2)$$

Assume the messages $a.m(0)$, $a.m(1)$, $a.m(1)$ are sent at different points in the future. Depending on how we define the meaning of the actor service, we may get different guarantees. One interpretation could be that for every pair of different messages of the form $a.m(x_1)$ and $a.m(x_2)$ that are received, a response message $b.m'(x_1 + x_2)$ is sent. This means in our case that $b.m'(1)$, $b.m'(1)$, $b.m'(2)$ are all sent eventually in some future state. However, in practice such guarantees are generally not given, because there would have to be some actor that keeps track of every message that was received until the the program finishes. In particular, one would not be able to capture the behaviour present in many programs such as the master's behaviour in Figure 14.

Instead, we require that an actor service with multiple triggers is associated with sessions governed by some protocol in an actor. For example, the given actor service may be associated with the sessions governed by protocol description ρ in actor a. We then write the actor service as

$$a.m(\underline{x_1}) \times a.m(\underline{x_2}) \rightsquigarrow^{(\rho,a)} b.m'(x_1 + x_2)$$

We require that the precondition of m specifies the session predicate $\rho(a)$. Hence we know that whenever a message $a.m(x)$ is sent, the corresponding session is running and will not be progressed until the message is received. The meaning of this actor service is the following. Whenever two different messages $a.m(x_1)$, $a.m(x_2)$ are received and it is guaranteed that they are received while the *same session* associated with (ρ, a) is running, then the response $b.m'(x_1 + x_2)$ will eventually be sent. By same session we mean that the session identifier of those two sessions are the same (there can be different sessions associated with (ρ, a)).

Consider the master in Figure 14. The master guarantees a sol message is sent whenever it receives two subres messages from the *same session*. Therefore the following actor service holds in the master

$$\underline{M}.\mathsf{subres}(\underline{r_1}) \times M.\mathsf{subres}(\underline{r_2}) \rightsquigarrow^{(SMR,M)} \exists C.\ C.\mathsf{sol}(r_1 + r_2) \tag{5.4}$$

We now give the general meaning of an actor service and the conditions that must be satisfied. Consider

$$T_1 \times ... \times T_k \rightsquigarrow^{\mathcal{S}} R$$

for trigger messages $T_1, ..., T_k$, response pattern R and session association $\mathcal{S}$. Assume $k > 1$ (more than one trigger message). The meaning of this actor service is that whenever trigger messages T_i are received, while the same session associated with $\mathcal{S}$ is running, then eventually after the reception of all trigger messages, one of the possible *complete responses* specified in R will be sent. This means all the response messages in the complete response will be sent (all the sent messages are guaranteed to be different). All the response messages are sent in different response states (all of which are strictly after each of the trigger states) and may be sent in any order. If there is only a single trigger message ($k = 1$), then there is no session association and the meaning of the actor service is analogous to the actor services in the original logic (except that multiple responses can be specified in the response pattern).

All one-state expressions in the response messages are evaluated in the corresponding response state. Hence each one-state expression in the where-clauses must be framed by the corresponding precondition or by an immutability predicate in the where-clause. For example, consider the following actor service

$$x.m() \rightsquigarrow x.f.n() \times x.f.n()$$

This actor specifies that whenever actor x receives message m, then eventually message n will be sent to $x.f$ in some future state and *another message* message n will be sent to $x.f$ in another future state. Even though the same expression $x.f$ is used for both response messages, $x.f$ may evaluate to a different value in each of those states.

If we have multiple trigger messages, it is not clear in general which state old expressions contained in the where-clause are evaluated in. To make this clear, we only permit the old expression $\textbf{old}(\iota_\rho(a))$ in these clauses (if there are multiple trigger messages) where the actor service is associated with sessions governed by ρ in a. This expression evaluates to the same value in each trigger state by definition of the actor service, since responses are only guaranteed, if the trigger messages are sent in the same session.

Furthermore, all the expressions in the trigger messages as well as the actor specified in the session association are evaluated in the current state (in the state where the actor service holds). We require that each of the trigger preconditions holds a session predicate associated with $\mathcal{S}$. If we did not require this, then there would be no easy way of showing that two or more trigger messages are received during the same session.

5.2 Sessions with join

In this section we extend the protocol descriptions and sessions introduced in Section 4 to be able to support join operations during sessions.

5.2.1 Protocol description

We add the notion of special join protocol states that a protocol description can define in addition to the usual protocol states. For each join state s in protocol description ρ one must specify

- The number of messages that are to be joined. This must be some statically known constant. We call this value the *multiplicity* $\mathcal{M}_{\rho,s}$ of the join.
- A join invariant $Inv^J_{\rho,s}(n)$ is a function from integers n to self-framing assertions. $Inv^J_{\rho,s}(n)$ holds, if the session is in join state s and n many join messages still must be joined. Hence we require that $Inv_\rho(s) = Inv^J_{\rho,s}(\mathcal{M}_{\rho,s})$ (recall that $Inv_\rho(s)$ is the assertion mapped to by the protocol invariant in protocol state s).

One may move to such a join state the same way as any other state. However, progressing the session or finishing the session from this state is done differently (see Section 5.2.3). In particular, the session state remains the same during the join and one may assume the join invariant at the beginning of a message handler for some n, if the session is in such a join state (instead of assuming the protocol invariant).

We define a protocol description *SMR* for the master actor in Figure 14. It has the following transition relation

$$\mathbf{QA} \sqsubset_{SMR} \mathbf{RS}$$

where the queryA message is expected in session state **QA** and the subres messages are expected in session state **RS**. **RS** is a join state with $\mathcal{M}_{SMR,\mathbf{RS}} = 2$ (since only 2 subres messages are expected). The protocol invariant is given by

$$\begin{aligned} Inv_{SMR}(s) := {} & acc(this.state) * acc(this.counter) * acc(this.result) * \\ & s = \mathbf{QA} \Rightarrow this.state = \mathsf{ACCEPTQUERYA} * \\ & s = \mathbf{RS} \Rightarrow \\ & \begin{pmatrix} this.state = \mathsf{WAITFORSUBRES} * \\ this.counter = 0 * this.result = 0 \end{pmatrix} \end{aligned}$$

The join invariant for **RS** is given by

$$\begin{aligned} \mathit{Inv}^{J}_{\mathit{SMR},\mathbf{RS}}(n) := {} & \mathit{acc}(\mathit{this.state}) * \mathit{acc}(\mathit{this.counter}) * \mathit{acc}(\mathit{this.result}) * \\ & \mathit{this.state} = \mathsf{WAITFORSUBRES} * \\ & \mathit{this.counter} = 2 - n * (n = 2 \Rightarrow \mathit{this.result} = 0) \end{aligned}$$

We have $\mathit{Inv}_{\mathit{SMR}}(\mathbf{RS}) = \mathit{Inv}^{J}_{\mathit{SMR},\mathbf{RS}}(\mathcal{M}_{\mathit{SMR},\mathbf{RS}})$.

5.2.2 Session ownership

In the original approach presented in Section 4 we can only give up partial session ownership (that is required to progress a session to one actor), since there exists only a single session predicate (see Section 4.2.2). However, at the fork we need to give up this ownership to multiple actors simultaneously, so that they can perform their computation and send the partial results over the subres messages in parallel. Therefore, whenever one moves to (or starts in) a join state, one gets hold of as many session predicates as is specified by the multiplicity. If actor a is controlling the session and moves to join state s, then one gets hold of $\underbrace{\rho(a) * \cdots * \rho(a)}_{\mathcal{M}_{\rho,s} \text{ many times}} * (\mathit{state}_{\rho}(a) = s)$. Each of these session predicates is required to progress or finish a session, hence they frame the same expressions as in the original approach.

A consequence of having session states where there are multiple session predicates means that we cannot always treat session predicates as exclusive permission as we did in the original approach (see Section 4.2.2). Instead, we only treat session predicates as exclusive permission, if one can show that the session state is not a join state. If the session state is a join state, then we treat session predicates as partial permission (only the actor doing the join knows how many session predicates there are).

5.2.3 Session progress

If a message handler is associated with a protocol description ρ, then in the original approach whenever it was invoked, one could assume the protocol invariant at the session state and then had to progress or finish the session (see Section 4.2.4). With join states this is slightly different. If the session state is a join state s, then one may assume the corresponding join invariant $\mathit{Inv}^{J}_{\rho,s}(n)$ for some initially unknown value n where $1 \leq n \leq \mathcal{M}_{\rho,s}$. If n is greater than 1, then the current message is not the last message to be joined, hence one needs to give up the received session predicate and ensure $\mathit{Inv}^{J}_{\rho,s}(n - 1)$. If n is 1, then the current message is the last message to be joined and one gets hold of all the given up session predicates ($\mathcal{M}_{\rho,s} - 1$ many) and one must either progress the session to a session state later in the transition relation or finish the session.

In the example given in Figure 14 the subres message is only received in the join session state **RS**. Hence at the beginning one may assume the corresponding join invariant given in Section 5.2.1 for some n. Only the non-failing branch is taken in the message handler due to the precondition and join invariant. The counter field is incremented by one and it is checked whether the incremented value equals 2. If this is the case, then one knows that $n = 1$, because the join invariant states that counter at the beginning of the message handler evaluates to n. Hence all the messages have been joined and one gets hold of the given up session predicate with which one can finish and start the session again.

If the incremented value of counter does not equal 2, then one knows that $n \neq 1$. At this point one can re-establish the join invariant for $n-1$, because one has incremented counter by 1 and one knows that $n \leq \mathcal{M}_{\rho,\mathbf{RS}} = 2$ which means $n-1$ is less than 2. Additionally, one must give up the session predicate at this point. We extend the validity of message handlers to include this extended definition of session progress in the case of join session states.

Note that it is sound to assume the join invariant at the beginning of the message handler. The reason is that the join invariant can only be assumed in message handlers that are associated with the protocol description and where it can be shown inside the body that it is currently in the join session state. Furthermore, in such a case the join invariant must be re-established at the end if the session is not progressed or finished. In no other case can one violate the join invariant, since the join invariant is self-framing, i.e. it holds all the permissions for the assertion that it maps to and if one cannot get hold of the join invariant, then one cannot get hold of these permissions.

5.2.4 Session events and receive event assertions

Recall that a session event (see Section 4.4) (ρ, a, i, s, m) for the session governed by ρ in actor a with session identifier i, describes the point in the session when message m is received in session state s. If s is not a join state, then this session event is a well-defined point in a session that can occur at most once. This why we were able to use it for environment expressions. However, if s is a join state, then this is not a well-defined point anymore since message m is received multiple times in the same session state for the same session. This is the reason why we only permit environment expressions that are associated with session events specifying non-join states.

Another implication of this fact is that receive event assertions (see Section 4.8) for such session events must be treated differently as well. A receive event assertion specifies a session event and if it holds, then this means that the session event has occurred. In the original approach we generated the receive event at the beginning of the message handler. For join protocol states we define receive event assertions as follows. A receive event assertion for a join session event holds if all the messages related to the join session event have been received. Hence we generate the corresponding receive event assertion when the last message is joined. Send event permissions however, can be treated the same as before (each session predicate for the join state can be transformed into the corresponding send permission).

This change enables us to use receive event assertions and send event permissions for join session events in interaction permissions. For example, an actor a that forks multiple messages to worker actors can accept interactions requested by the workers which specify these join events. An example interaction permission specifying an interaction from the point of view of a with one of the workers (worker1) could be

$$\textsc{rcv}(E)\ .\ \textsc{send}_{\rho_W}(\mathsf{worker1}, \iota_{\rho_W}(\mathsf{worker1}), s, \mathsf{compute})\ .\ \textsc{EndS}$$

where E is some join session event in the actor a. worker1 would be obliged to send the send permission for the continuation of this interaction permission to get hold of $\textsc{send}(E)$ which it would require to send the join message. a, who is performing the join, would only get hold of the send permission in the continuation if it got hold of the receive event assertion for E. It would only get hold of the receive event assertion, if it joined all messages, at which point it is guaranteed that worker1 has sent its send permission.

The advantage of being able to use interaction permissions for join states in this example is that one need not specify each of the required send permissions in the preconditions of each of the messages that are exchanged between a and the workers, which would require exposing implementation details as discussed in Section 4.7. Instead, the requested interaction permission takes care of the expected transfer of send permissions.

5.3 Proving local actor services

As presented in Section 2.7, in the original actor services logic a local actor service is an actor service that can be proved by just examining the trigger message handler implementation, i.e. the corresponding response is sent in the trigger message handler. For the generalized actor services we extend local actor services to be actor services that can be proved by just examining all the trigger message handler implementations (which must be part of the same actor), i.e. the corresponding response is sent in one of the trigger message handlers.

For the sake of presentation, if a local actor service has more than one trigger message, then we require that all the handlers of the trigger messages are the same. For example, we support local actor services such as

$$\underline{a}.m_1(\underline{x_1}) \times a.m_1(\underline{x_2}) \rightsquigarrow R$$

but we do not support local actor services such as

$$\underline{a}.m_1(\underline{x_1}) \times a.m_2(\underline{x_2}, \underline{x_3}) \rightsquigarrow R$$

We show how to lift this restriction in Section 5.3.2, but it requires more technical details that distract from the main idea. As we will see in Section 5.5.4 when showing how to compose such general actor services, one can still derive non-local actor services that have different trigger messages with this restriction.

5.3.1 Proving local actor services with multiple triggers

To prove a local actor service $\underline{a}.m(\underline{x_1}) \times ... \times a.m(\underline{x_k}) \rightsquigarrow R$ we require that

- Message handler m is associated with a protocol ρ and its precondition makes sure that it can only be sent in one particular join session state s.
- The number of trigger messages must be equal to $\mathcal{M}_{\rho,s}$.

The restriction that it must be possible to send message m in a join state should be clear from our previous discussion. During the join the actor keeps track of various information in its local state and we need to capture this. The way we do this is to associate the join to a session using the extended protocols and sessions from Section 5.2.

The motivation for the restriction that the number of trigger messages equals the multiplicity of a particular join state also is clear, because we know that the response will only occur once all the messages are joined. However, the restriction that m can only be invoked in one particular join state is not intuitive and is a result of how our actor services are currently defined. Suppose we did not have this restriction, then we would have to be able to differentiate in the response pattern of the actor service, if each of the trigger messages is invoked in the corresponding join state or not. Currently, the only way to talk about the trigger states is using old expressions in where-clauses, which are not expressive enough to make this differentiation due to the restriction imposed on old expressions with multiple trigger messages (see Section 5.1). We leave such an extension for future work.

Next, we explain the main rule for proving a local actor service

$$\underline{a}.m(\underline{x_1}) \times ... \times a.m(\underline{x_k}) \rightsquigarrow e.m'(\overrightarrow{e_i}) \textit{ where } A$$

for all actors a of some actor type under these restrictions. We will generalize to arbitrary response patterns later. We illustrate the different techniques also by showing what one would have to do to prove the master's local actor service (5.4). Suppose m is associated with ρ and it is always sent in join state s. We first need to prove that m is a valid message handler as described in Section 5.2.3. This shows that the join progresses each time message handler m is invoked using the join invariant $Inv^J_{\rho,s}(n)$. However, this does not show how the actor's local state progresses with respect to the arguments provided by the different m messages.

In the example given in Figure 14 the master adds the value provided in each subres message to its result field. Since the subres messages may arrive in any order, we cannot precisely express what value result will have, if all we know is that one message has been joined (but we do not know which one). Therefore the join invariant cannot be easily used to track this information.

The main observation is the following. Each time a message handler is invoked that is used for the join, the local state tracking the computation performed by the join (which we refer to as the *join computation state*) is updated by some transforming effect dependent on the message handler's arguments. If we can capture this effect and we can show that it does not matter in which order the effect is applied with respect to the arguments of all the received messages, then we can express the final join computation state by applying this effect repeatedly to the different arguments *in any order*.

In our example the join computation state is given by the result field and the transforming effect is just the addition of the argument to this field. It does not matter in which order this effect is applied to the arguments since addition is associative and commutative. Repeatedly applying this effect to the arguments in any order will yield the sum of the arguments. More generally, let f be a binary function that captures the transforming effect. Assume the current values stored in the join computation state are given by $\overrightarrow{v_j}$, then $f(\overrightarrow{x_i}, \overrightarrow{v_j})$ gives the updated values once the join message handler has been invoked, where $\overrightarrow{x_i}$ are the arguments provided for this particular message (in our example $f(x, v) = x + v$). In particular, if the messages that are received for a particular join have arguments $\overrightarrow{x_k}, \overrightarrow{x_{k-1}}, ...\overrightarrow{x_1}$ (in the order received) then the final values of the join computation state is given by

$$f(x_1, f(x_2, ..., f(x_k, \overrightarrow{v_0})))$$

where $\overrightarrow{v_0}$ are the initial values when the join starts. We say that f is *order-independent* if the following holds

$$\forall \overrightarrow{x_i}, \overrightarrow{x_i'}, \overrightarrow{v_j}.\ f(\overrightarrow{x_i}, f(\overrightarrow{x_i'}, \overrightarrow{v_j})) = f(\overrightarrow{x_i'}, f(\overrightarrow{x_i}, \overrightarrow{v_j}))$$

If f is order-independent, then one can show that the final values computed at the end of the join will be the same independent of the order in which the messages arrive (because order-independence essentially permits moving any of the arguments $\overrightarrow{x_i}$ in the final expression to any other position). Hence the final value is given by the same expression as for the fixed order given before. Another common way of writing down this expression is using *folds*. We have that the final value is given by $foldr(f;\ \overrightarrow{v_0};\ (\overrightarrow{x_1}, ..., \overrightarrow{x_j}))$ where

$$foldr(f;\ \overrightarrow{v_0};\ (\overrightarrow{x_1}, ..., \overrightarrow{x_j})) = f(x_1, f(x_2, ..., f(x_k, \overrightarrow{v_0})))$$

We have now introduced the most important tools and can move on to the actual rule for proving the local actor service. One must pick one-state expressions $\overrightarrow{e_j}$ which make up the join computation state (this is the state that is required to track the computation of the join). We require the following

$$\forall n.(1 < n \leq \mathcal{M}_{\rho,s}) \Rightarrow Inv^J_{\rho,s}(n) \models_{frm} \overrightarrow{e_j}$$

which states that $\overrightarrow{e_j}$ must be framed by the join invariant for all feasible n. This is required to ensure that these expressions remain stable between invocations of message handler m. Next, one must pick one-state expressions $\overrightarrow{e_j^{init}}$ which evaluate to the initial values of $\overrightarrow{e_j}$ right when the join has started. We require that $\overrightarrow{e_j^{init}}$ is immutable during the session once the join starts. This means that all the subexpressions in $\overrightarrow{e_j^{init}}$ are either framed by immutability predicates in the join invariant $Inv^J_{\rho,s}(\mathcal{M}_{\rho,s})$ or mention the corresponding session identifier. The reason why mentioning the session identifier is permitted is that the session identifier stays the same during the complete session. We can ensure that those are the initial values by enforcing

$$Inv^J_{\rho,s}(\mathcal{M}_{\rho,m}) \models \overrightarrow{e_j} = \overrightarrow{e_j^{init}}$$

For the master, we have $\overrightarrow{e_j} :=$ **this**.result (which is framed by the join invariant) and $\overrightarrow{e_j^{init}} := 0$. We have that **this**.result $= 0$ when no messages have been joined yet. Furthermore, one must pick a binary function f that captures the transforming effect and is order-independent. For the master we pick the addition function.

At the beginning of message handler m one may assume $Inv^J_{\rho,s}(n)$ for some n (as in Section 5.2.3). Suppose $\overrightarrow{e_j}$ evaluates to values $\overrightarrow{v_j}$ at the beginning of the message handler with arguments $\overrightarrow{x_i}$. Hence $\overrightarrow{v_j}$ are the current values of the join computation state. If $n > 1$ (i.e. it is not last message to be joined), one must show at the end of the message handler that the join computation holds the updated values as prescribed by the transforming effect function f. So, one must show

$$f(\overrightarrow{x_i}, \overrightarrow{v_j}) = \overrightarrow{e_j}$$

at the end of the message handler. In our example one must show $r_i + v =$ **this**.result at the end, where v is the value of **this**.result at the beginning and r_i is the argument. This holds in the example, since **this**.result is incremented by r_i.

If $n = 1$ (it is the last message to be joined), one must show that the expected response is sent. This can be done using the existing machinery provided by the original actor services logic (at least for single response messages, we describe how to do this for multiple response messages in Section 5.3.3). Since we know that all the other messages have been joined, we may assume that applying the transforming effect to the join computation state will yield the final value. Hence we may assume at the beginning of the message handler

$$f(\overrightarrow{x_i}, \overrightarrow{v_j}) = foldr(f;\ \overrightarrow{v_j^{init}};\ (\overrightarrow{x_1}, ..., \overrightarrow{x_k}))$$

where $(\overrightarrow{x_1}, ..., \overrightarrow{x_k})$ are the formal arguments of all the message handlers in the trigger. $\overrightarrow{e_j^{init}}$ evaluates to $\overrightarrow{v_j^{init}}$ (i.e. the initial values since $\overrightarrow{e_j^{init}}$ is guaranteed to be immutable during the session).

In our example one may assume $r_1 + v = r_1 + r_2$ (where r_1, r_2 are the arguments of two subres messages, v is the value of **this**.result at the beginning; $\overrightarrow{v_j^{init}}$ drops out since it is given by 0). Since one sends a sol message with value $r_1 + v$, one can show that the message will contain the sum of the arguments. We conclude that the actor service (5.4) can be derived.

We note that in local actor services we can prove where-clauses the same as in the original logic. If the response is sent in a particular message handler (i.e. if the join has finished), then one must show that the where-clause relates the response state with the state at the beginning of the message handler. One may only refer to the corresponding session identifier in old expressions, which is guaranteed to evaluate to the same value in each of the trigger states.

5.3.2 Different trigger message handlers in local services

We have only shown how to deal with local actor services, where all the trigger message handlers are the same. Generalizing this to multiple trigger message handlers is possible. The main challenge is that the different trigger message handlers can have different effects on the join computation state. They might even have effects on disjoint parts of the state. In such situations it does not make sense to have a single session predicate that can be used to send any of the trigger messages involved in the join, since one requires a way to control how often which effect is applied. One could generalize this by having different types of session predicates during the join, each of which could only be used for one sort of message handler.

One then would require separate functions describing the different effects and one would have to define order-independence between functions, which is straightforward. The final result would then be computed as the composition of the different folds.

5.3.3 Proving local services with multiple responses

We saw in Section 5.3.1 how to prove local actor services with multiple triggers and a single response message. Extending this to arbitrary response patterns is straightforward. The original logic provides a way to show that a single response message is sent among some set of alternatives. One can generalize this by first having to select a complete response and then showing that all the response messages in the selected complete response are sent, which can be done analogously to the single response message case in the original logic.

5.4 A more precise local actor service for the master

We outlined in Section 5.3.1 how to derive local actor service (5.4) which describes the master's join behaviour but it is not precise enough, since it does not specify to which client it sends the solution. Instead, we can derive analogously the following local actor service that specifies the client

$$\begin{aligned} \underline{M}.\mathsf{subres}(\underline{r_1}) \times M.\mathsf{subres}(\underline{r_2}) \rightsquigarrow^{(SMR,M)} \exists C.\ C.\mathsf{sol}(r_1 + r_2) \\ \textit{where } C = \varphi_{SMR}(M, old(\iota_{SMR}(M)), \mathbf{QA}, \mathsf{query(c,n)}, \mathsf{c}) \end{aligned} \tag{5.5}$$

The where-clause is permitted, since $old(\iota_{SMR}(M))$ is the only old expression and the actor service is associated with the corresponding session.

In actor service (5.5) the client to whom the solution is sent is specified as the client that was received over the query message in the same session in session state **QA**. This can be achieved using environment expressions (see Section 4.4).

5.5 General composition

In Section 5.3 we showed how to prove local actor services with multiple triggers and multiple response messages. In this section we show how one can compose such actor services. We do this by presenting formal composition rules and by showing how to apply them in the example given in Figure 14. First, we introduce semantic judgements from the original actor services logic, which we require for the rules. Next, we present the two different composition rules in our setting.

5.5.1 Semantic judgements

In the semantics of the original actor services logic [20] the semantic judgement for one-state assertions (it does not mention **old**) has the form $\Lambda, \Sigma, \sigma \models a$, where

- Λ is an actor service environment holding the assumed local actor services (which hold in every program state).
- Σ is a heap-state that consists of a heap as well as a set of exclusive and immutable permissions.
- σ maps variables to values.
- a is a one-state assertion.

The interpretation of the semantic judgement is that if one assumes the actor services in Λ, then the assertion a holds in the program state with the heap and permissions specified by Σ and the variable mapping given by σ.

The semantic judgement for two-state assertions has the form $\Lambda, \Sigma_1, \Sigma_2, \sigma \models A$ where Σ_1, Σ_2 are heap-states and A is a two-state assertion. The interpretation of this semantic judgement is analogous to the one-state version. **old** expressions in A are evaluated in Σ_1, while other expressions are evaluated in Σ_2. Furthermore, the judgement $\Sigma, \sigma \models_{immut} e$ for a one-state expression e states that e is immutable in the corresponding state.

5.5.2 Basic composition

The first composition rule is almost identical to the composition rule in the original actor services logic. We motivate the rule using our example in

Figure 14. It is straightforward to derive the following local actor service for the master which expresses the fork behaviour

$$\begin{array}{l}\underline{M}.\mathsf{queryA}(C,n) \rightsquigarrow \exists W_a, W_b. \\ \quad (W_a.\mathsf{computeA}(M,n)\ \mathit{where}\ \mathit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M)) \times \\ \quad (W_b.\mathsf{computeB}(M,n)\ \mathit{where}\ \mathit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M))\end{array} \tag{5.6}$$

The composition rule permits composing *single* response messages the same way as in the original actor services logic. So we can evolve the computeA response message independently from the computeB response message. This is fine, because the response states of the different response messages need not be related in any way. We can derive the local actor service (5.2) describing the reaction of WorkerA to the computeA message. We compose the computeA branch in (5.6) with (5.2) to get

$$\begin{array}{l}\underline{M}.\mathsf{queryA}(C,n) \rightsquigarrow \exists W_b. \\ \quad (M.\mathsf{subres}(f(n))\ \mathit{where}\ \mathit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M)) \times \\ \quad (W_b.\mathsf{computeB}(M,n)\ \mathit{where}\ \mathit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M))\end{array} \tag{5.7}$$

This composition step is completely analogous to the original composition, the only difference is that we had to pick which response message to compose. The formal rule describing such compositions is given by[13]

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

c ranges over complete responses (i.e. sets of response messages) and R, R', R'' over response patterns (i.e. sets of complete responses). $(\mathit{map}\ (\cup c)\ R'')$ denotes the response pattern where the response messages specified by c are added to all the complete responses in R''.

The composition requires that one can show that right before the response message (this state is related to the trigger state using A) is sent, it is guaranteed that the actor service describing the reaction to the response holds. This is done using $(\Lambda, \Sigma, \sigma).\mathit{futureEntails}(A_1, A_2)$ which checks entailment between assertions A_1 and A_2 in all pairs of future heap states Σ_1, Σ_2 (where $\Sigma \prec \Sigma_1 \prec \Sigma_2$). The final premise

$$(\Lambda, \Sigma, \sigma).\mathit{futureCombines}(A, R', R'')$$

[13] We omit explicit treatment of existential variables for the sake of presentation.

is required to make sure that the newly introduced where-clauses in the resulting actor service relate the correct trigger and response states. The ideas are the following

1. A describes the relation between the heap state Σ_1 when T occurs[14] and the response state Σ_2 when $e.m(\overrightarrow{e_i})$ is sent. Since all expressions in A evaluated in the response state are framed by the precondition of m (or are immutable), we may assume that the expressions evaluate to the same values in the state Σ_2' when $e.m(\overrightarrow{e_i})$ is received. Hence A also describes the relation between Σ_1 and Σ_2'.

2. Every where clause in R' describes the relation between the state Σ_2' when $e.m(\overrightarrow{e_i})$ is received and the state Σ_3 when the corresponding response is sent.

3. R'' has the same response pattern as R', except for the where-clauses. Under the assumptions of the first two points one must make sure that each where-clause in A describes the relation between Σ_1 (when T occurs) and Σ_3 (when the response is sent)

The *futureCombines* relation makes sure that R'' satisfies the condition given the assumptions on A and R'. Its precise definition is given in [20].

We can analogously compose the other branch in (5.7) with the local actor service (5.3) to get

$$\begin{aligned}&\underline{M}.\mathsf{queryA}(C, n) \rightsquigarrow \\ &\quad (M.\mathsf{subres}(f(n))\ \textit{where}\ \textit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M)) \times \\ &\quad (M.\mathsf{subres}(g(n))\ \textit{where}\ \textit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M))\end{aligned} \tag{5.8}$$

Now both branches have evolved. We have shown that once the master gets a queryA message, then eventually it will receive the two partial results. We can now apply the rewrite rule introduced in Section 4.5 to rewrite the $M.\mathsf{subres}(f(n))$ branch. Again the rule is the same, except that we need to

[14] If T specifies multiple trigger messages, then A can only talk about the session identifier in the trigger states. In this case one can think of Σ_1 as the state when the last trigger message is received. In this case R'' can also only talk about the session identifier in the trigger states.

pick a response message[15]. We get

$$\begin{array}{l}\underline{M}.\mathsf{queryA}(C,n) \rightsquigarrow \\ \left(\begin{array}{l} M.\mathsf{subres}(f(n))\ \mathit{where} \\ \quad \mathit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M)) * \\ \quad C = \varphi_{SMR}(M, \mathit{old}(\iota_{SMR}(M)), \mathbf{QA}, \mathsf{query(c,n)}, \mathsf{c}) \end{array}\right) \times \\ \left(M.\mathsf{subres}(g(n))\ \mathit{where}\ \mathit{old}(\iota_{SMR}(M)) = \iota_{SMR}(M) \right)\end{array} \tag{5.9}$$

5.5.3 Join composition

In the previous section we only composed single response messages, which is analogous to the composition in the original actor services logic. Now, we introduce the second composition rule, which allows the composition of multiple response messages with multiple trigger messages simultaneously, which is fundamentally different to the composition in the original actor services logic. The rule is given by (the session association in the conclusion can be removed if $T' = \emptyset$)

$$\frac{\begin{array}{l} \Lambda, \Sigma, \sigma \models e.m(\overrightarrow{e_i}) \rightsquigarrow \left\{\left\{ \begin{array}{l} (e_1.m(\overrightarrow{e_{i_1}})\ \mathit{where}\ A_1), ..., \\ (e_k.m(\overrightarrow{e_{i_k}})\ \mathit{where}\ A_k) \end{array} \right\} \cup c \right\} \cup R \\ \left(\begin{array}{l} \forall j \in \{1,2,...,k\}. \\ (\Lambda,\Sigma,\sigma).\mathit{futureEntails}\left(\begin{array}{l} A_j, \\ \left(\left\{ \begin{array}{l} e_1.m(\overrightarrow{e_{i_1}}), ..., \\ e_k.m(\overrightarrow{e_{i_k}}) \end{array} \right\} \cup T' \right) \rightsquigarrow^{(\rho, e_a)} R' \end{array} \right) \end{array} \right) \\ \Sigma, \sigma \models_{\mathit{immut}} e_a \\ X, \overrightarrow{X_i}, Z \notin \mathit{dom}(\sigma) \quad \sigma' = \sigma[X \mapsto \llbracket e \rrbracket_{\Sigma,\sigma}][\overrightarrow{X_i} \mapsto \llbracket \overrightarrow{e_i} \rrbracket_{\Sigma,\sigma}][Z \mapsto \llbracket e_a \rrbracket_{\Sigma,\sigma}] \\ \forall \Sigma'.\ \Sigma', \sigma' \models \mathit{pre}(m, X, \overrightarrow{X_i}) \Rightarrow \Sigma', \sigma' \models \rho(Z) \\ \forall j \in \{1,2,...,k\}.\ (\Lambda,\Sigma,\sigma).\mathit{futureEntails}(A_j, \mathit{old}(\iota_\rho(e_a)) = \iota_\rho(e_a)) \\ \forall j \in \{1,2,...,k\}.\ (\Lambda,\Sigma,\sigma).\mathit{futureSimmut}((\rho, e_a), A_j, A'_j) \\ (\Lambda,\Sigma,\sigma).\mathit{futureCombines}(A'_1 * A'_2 * \cdots * A'_k, R', R'') \\ T' \neq \emptyset \Rightarrow (R = \emptyset \wedge c = \emptyset) \end{array}}{\Lambda, \Sigma, \sigma \models (\{e.m(\overrightarrow{e_i})\} \cup T') \rightsquigarrow^{(\rho, e_a)} (\mathit{map}\ (\cup c)\ R'') \cup R}$$

The rule enables the composition of two actor services. The first actor service (we refer to this actor service as the fork actor service) has a single trigger $e.m(\overrightarrow{e_i})$ and multiple response messages $e_1.m(\overrightarrow{e_{i_1}})$,...,$e_k.m(\overrightarrow{e_{i_k}})$ which are all part of the same complete response. The second actor service (we refer to

[15]The rule can only be applied if we have a single trigger message, because one cannot refer to arbitrary old expression with multiple triggers. Furthermore, one must guarantee that the generated environment expression is well-defined (i.e. the session state referred to cannot be a join state, see Section 5.2.4).

this actor service as the join actor service) has multiple trigger messages of which a subset must match the response messages of the fork actor service.

Let us analyze the premises of the rule. One must first show that the fork actor service holds in the current state. Next, one must show that the join actor service holds before the response messages, which we want to compose, are sent. This is ensured by checking that the join actor service holds in each response state using the premise

$$\begin{aligned}&\forall j \in \{1, 2, ..., k\}.\\ &\quad (\Lambda, \Sigma, \sigma).\mathit{futureEntails}\left(\begin{array}{l} A_j, \\ \left(\{e_1.m(\overrightarrow{e_{i_1}}), ..., e_k.m(\overrightarrow{e_{i_k}})\} \cup T' \right) \rightsquigarrow^{(\rho, e_a)} R' \end{array} \right)\end{aligned}$$

The intuition for this premise is that since the response messages may be sent in any order we show that independent of which response message is sent first, we can ensure that the join actor service will hold[16]. If one has shown this, then one knows that the join actor service is associated with sessions (ρ, e_a) where e_a is an expression that evaluates to an actor. To ensure that we can talk about this actor in the fork actor service, we require that in the current state e_a is immutable, guaranteeing that e_a evaluates to the same actor as the join actor service specifies.

At this point even though we know that the response messages sent in the fork are all different and match a subset of the trigger messages in the join actor service, we do not have enough information to use the guarantees provided by the join actor service. The join actor service only guarantees the response if the trigger messages are received during the same session associated with (ρ, e_a). To show that they are received during the same session, one needs to prove that in each response state one can show that the session identifier when the response is sent is the same as when the trigger of the fork actor service is received. This ensures that in all the response states when the response messages are sent, the session identifier is the same.

Furthermore, to make sure one can even talk about the session identifier in the trigger message, we require that its precondition holds the corresponding session predicate. Since the precondition of the join actor service must require the corresponding session predicate that it is associated with, we can conclude that we can talk about the session identifier in the response messages of the fork actor service. Furthermore, this also means that the session cannot progress between the sending of the response messages and their reception.

[16]We use here that in the original logic to derive an actor service in every future state with respect to current state Σ, one must show that all heap-dependent expressions that appear in the trigger and response of the actor service are immutable in Σ. This ensures that in each response state we are talking about the same join actor service.

Hence it is guaranteed that all the response messages are received in the same session.

Next, we must make sure that in the composed actor service the where-clauses are adjusted appropriately. The intuition for this premise (which involves the *futureCombines* relation) can be explained as follows. Let Σ_1 be the state when the trigger message $e.m(\overrightarrow{e_i})$ is received. Let Σ_2 be the state when the final fork response message is sent. Take any where-clause A_i describing the relation between Σ_1 and the response state when the corresponding fork response message is sent. Now let A'_i be an assertion that is entailed by A_i and which only includes old expressions, one-state expressions which are immutable or which are given by $\iota_\rho(e_a)$. A'_i relates Σ_1 and Σ_2, because the immutable expressions still have the same value at Σ_2 and we know that the session identifier when all response messages are sent is the same. Also, old expressions are always is evaluated in Σ_1.

Now take any where-clause A_J in the response pattern of the join actor service. Let Σ_3 be the state when the corresponding response message is sent. We know that Σ_3 is in the future of Σ_2 since the response message in the join actor service is only sent after the last fork response message has been received. We know that the only old expression A_J can refer to is **old**$(\iota_\rho(e_a))$, which we know evaluates to the session identifier when the trigger messages are received. Since the trigger messages hold the corresponding session predicates, we know that when the last response message is received by the join actor service the session identifier is the same as when it was sent (i.e. the same as in Σ_2). Furthermore, since Σ_3 is in the future of Σ_2 we know that any expression immutable at Σ_2 must also be immutable at Σ_3 when the response message is sent. We conclude that A_J relates Σ_2 and Σ_3.

Hence we may transform all where-clauses A_i in the fork actor service to A'_i (only holding immutable expressions and the session identifier expression, this relation between A_i and A'_i is guaranteed by the *futureSimmut* relation) and use all these facts together to relate Σ_1 and Σ_2. Using the *futureCombines* relation we can then ensure that each where-clause in the resulting response pattern R'' relates the state when the trigger message $e.m(\overrightarrow{e_i})$ is received (Σ_1) and when the join response message is sent (Σ_3). This exactly what we want if $T' = \emptyset$.

If $T' \neq \emptyset$ then the resulting actor service has multiple triggers. This means that each where-clause in R'' can only refer to old expression **old**$(\iota_\rho(e_a))$ and hence the only facts that were carried into these where-clauses about Σ_1 involve the session identifier and no other expression evaluated in Σ_1. However, since the session identifier is the same in Σ_1 and Σ_2, these facts are guaranteed to hold in Σ_2 as well. We conclude that the where-clauses in R'' in this case still relates the state when the last response message is received (this state may be at a different point now, since some of the trigger messages have changed) and when the join response message is sent.

Finally, premise $T' \neq \emptyset \Rightarrow (R = \emptyset \wedge c = \emptyset)$ ensures that if the resulting actor service has multiple triggers, then all the responses in the response patterns are guaranteed to occur after all the trigger messages have been received. If the resulting actor service has a single trigger, then this is guaranteed directly.

Using this rule we can compose the actor service (5.9) with the master's join behaviour expressed using local actor service (5.5)

$$\underline{M}.\mathsf{queryA}(\underline{C}, \underline{n}) \rightsquigarrow C.\mathsf{sol}(f(n) + g(n))$$

which is what we wanted to derive. Let us make sure that the premises of the rule are satisfied. The join actor service in this case is a local actor service and since all the parameters in the corresponding fork actor service are universally quantified, we can show the correct join actor service in every future state[17]. The actor e_a specified in the session association is the master M that is universally quantified, so we can show immutability of this expression. We can show in both response messages in the fork actor service that the session identifier does not change. We can use the fact $C = \varphi_{SMR}(M, \iota_{SMR}(M), \mathbf{QA}, \mathsf{query(c,n)}, \mathsf{c})$ in one of the fork responses since C, M are universally quantified and other than those expressions it only refers to the session identifier expression $\iota_{SMR}(M)$.

5.5.4 Actor services with multiple different triggers

Until now we only derived local actor services with multiple triggers which always have the same trigger message multiple times (at least with our restriction). Using the introduced composition rule, we can derive actor services which have different trigger messages. For example, we may compose the reaction of WorkerA given by actor service (5.2) with an instantiation of actor service (5.5) describing the master's join behaviour to get

$$\begin{aligned} \underline{W}.\mathsf{compute}(\underline{M}, \underline{a}) \times M.\mathsf{subres}(\underline{r_2}) &\rightsquigarrow^{(SMR,M)} \exists C.\ C.\mathsf{sol}(f(a) + r_2) \\ \textit{where } C &= \varphi_{SMR}(M, \textit{old}(\iota_{SMR}(M)), \mathbf{QA}, \mathsf{query(c,n)}, \mathsf{c}) \end{aligned} \tag{5.10}$$

We have now derived an actor service which has different trigger messages. We could continue this approach by composing the other worker's behaviour as well and then composing the resulting actor service with the initial fork

[17] In the actual derivation one must explicitly deal with the quantified variables. We do not show this here and refer to [20] for details.

local actor service (5.6) of the master. This would result in the actor service describing the main response property, but where the derivation followed a different order compared to the derivation shown in Section 5.5.3.

5.6 Concluding remarks

We extended actor services to support multiple triggers and multiple responses, allowing to describe the different parts of a fork-join behaviour separately and to then compose these behaviours modularly. The notion of a session introduced in Section 4 was vital for this extension. However, as we have pointed out, there are several restrictions that we impose. One important restriction that we did not cover is the following. If we compose multiple response messages simultaneously of an actor service $T \rightsquigarrow R$ with another actor service, then we require that T only contains a single trigger message. The main reason is that the only way for us to show that the different response messages are associated with the same session is by relating them to the trigger state. This is only possible in general if there is only one such trigger state. It would be interesting to investigate if there is a way to ensure more directly that multiple branches are associated with the same session. Furthermore, making the old expressions in actor services with multiple triggers less restrictive could be helpful as well.

Another important extension of the current approach would be to permit join multiplicities that are unbounded. The main challenge is that one needs a way to keep track of this value somehow and in the composition one needs to show that the number of messages forked matches this value.

6 Related work

6.1 Session types

The general notion of a *session* as motivated in Section 4 is well-known. In process calculi channels can be used as ports between two communication partners (using the two channel endpoints) over which data is sent. Channel endpoints themselves can be passed as data over other channels (channel delegation, i.e. the communication partners are not fixed). In this context a session is defined as the interaction that occurs over the channel from the creation of the channel until its disposal. *Binary session types* [11] to describe the sequence of events of such a session from the point of view of one channel endpoint are introduced by Honda *et al.*. The two binary session types for the two endpoints of the same channel are *dual* to each other. A type system is used to ensure that a channel endpoint is used according to its session type.

One can interpret protocol descriptions (see Section 4.2.1) in our setting as some kind of session type, however there are many differences. Protocol descriptions do not specify in which state the session must start or finish. Also one may move to any session state that comes later in the ordering instead of just the next states in the ordering. Furthermore, protocol descriptions are decoupled from the actual events. This association between session states and message handlers is done independently. Also one does not have two independent endpoints. Protocol descriptions only describe the type of the session for the actor controlling the session. It is not possible for an actor to send one message followed directly by another message for the same session without the receiving actor reacting (this is a direct consequence of no message ordering being assumed, see Section 2.2.1).

The duality defined for interaction permissions (see Section 4.8) is analogous to the duality of two channel endpoints. In fact one can think of the two companion permissions a bit like two channel endpoints. Send event permissions correspond to output capabilities and receive event assertions correspond to input capabilities. One major difference is that progression of a interaction permission is fundamentally different from progressing channel, since one must provide permissions or show that an assertion holds. The permission that is given up to progress one interaction permission, can be obtained by the companion interaction permission. However, no data is sent between the interaction permissions. In some sense the permission that is transferred over interaction permissions is similar to the data sent over channels. Furthermore, creation of an interaction permission is different as well, since it requires an explicit handshake between two actors agreeing on some interaction over a request.

Honda *et al.* [12] extend the notion of binary session types to multiparty session types to describe interactions involving more than two parties, where parties communicate over asynchronous messages (with more ordering guarantees than in our setting). A multiparty session type (or *global type*) can be projected to a *local type* to obtain the point of view of a single participant in the interaction.

A fundamental difference of our work compared to multiparty session types is that we are interested in proving response properties, while in the session types work one is interested in showing that a program behaves according to a global protocol. Nevertheless, there are connections to our work.

In our setting having a global type for the whole program is infeasible, since this requires knowing the complete program and hence does not facilitate modularity. While transferring permissions by progressing interaction permissions is similar to sending data over channels in the binary session type work, the interaction that an interaction permission specifies in a request clause is a bit like a projected local type describing the interaction from the point of view of the actor making the request. One difference is that such a requested interaction may only describe part of the projected type with respect to the complete global type. Furthermore, we never check if an actor follows the complete interaction specified in the request, we only make sure that if the corresponding interaction permission is progressed, then the interaction can continue to the next step as described.

In some sense we try to derive response properties about larger parts of the program using just the projected types. Lange and Tuosto [13] develop an approach to synthesise local types to construct the global type which is similar in terms of the idea (going from local to more global). The synthesised global types, however, cannot be synthesised further, making it unsuitable for our work. Nevertheless, it would be interesting to see if one could use such a synthesis to describe global types that just summarize part of the actor program and which one could compose further to get more global types. This could be useful in the examples discussed in Section 4.11.

6.2 Actors and session types

There have been different works that have incorporated session types to reason about actor programs. In [14] Mostrous and Vasconcelos introduce session types for a subset of Erlang. Their type system can be used to show that two actors communicate according to some binary protocol. As in the session types work this goal is different from ours.

They support parallel sessions using correlation sets, which permits an actor to have multiple sessions of the same protocol type simultaneously. Furthermore, they permit multiple session initialization messages to be sent simultaneously (which we do not). However, their type system only makes guarantees once such an initialization message is received (hence they cannot provide liveness guarantees).

In our setting a session is described in terms of a single actor and during this session the actor can communicate with many different actors (not just a single one). Hence they do not support programs such as the one presented in Section 4.7. They do not support delegation of the session. In our setting we can support delegation, since one can give up the send permission to another actor that then sends the actual message associated with the send permission.

Neykova and Yoshida propose multiparty session actors [15]. This is a generalization of multiparty session types to the actor setting. A given global type is checked to be followed by the program at runtime. Fowler [9] builds on this work by providing an implementation for Erlang and makes other extensions. One difference to our work (apart from their goal being about the conformance to a global type) is that it relies on runtime verification. In their setting they do not require actors to know the identities of the actors they are sending messages to. Instead, the actors can use the name of the role of the receiving actor in the protocol and the runtime system makes sure the message goes to the right actor. This is fundamentally different to the assumptions in our work. As in the multiparty session types work, the projected local types of the actors have some connection to interactions requested by actors in request clauses in our setting.

6.3 Mailbox calculus

De'Liguoro and Padovani [8] propose the mailbox calculus and a corresponding type system to reason about actor programs. One of the main ideas is to make the mailbox that an actor can contain a first order citizen in the calculus, which means one can pass around mailboxes, which our actor language does not support. Their type system types mailboxes. A mailbox can either be typed using an *input capability type* or an *output capability type*. An input capability type describes what messages one must expect next and an output capability describes what messages must be sent to the mailbox next. So if one owns a mailbox that is typed using an output capability that states one must send message m_1 next, then sending message m_2 to this mailbox will not type check. There can only be one input capability in the system at any given point, but there can be multiple output capabilities.

The output capabilities are similar to our session predicates (or send event permissions), since they represent permission to send a message. One difference is that we do not support giving out multiple session predicates (except in the fork-join case). However, using interaction permissions we setup a single interaction that ensures that upon receiving different messages an actor can receive different send event permissions with respect to the same actor reference that is part of the same session. To the best of our knowledge, such a signal that indicates at which point one gets an output capability cannot be done in the mailbox type system. One needs to get hold of another reference to the same actor which has another type, which in our setting is similar to specifying all the different session predicates that must be transferred over messages in the corresponding preconditions, which as we saw in Section 4.7 exposes implementation details.

6.4 Other related work

Conversation types [6] describe conversations in a multiparty setting (there are some similarities to multiparty session types). It permits a participant to delegate part of the conversation to another entity, while keeping another part of the conversation to itself. This has some similarities to our setting. We enable an actor controlling a session to request multiple interactions with other actors that gain access to different parts of the session during different points, while the actor controlling the session takes part in each of the interactions. Conversation types are less restrictive than in our case, however, they require a global view of the conversation, which is not modular.

In [17] Padovani *et al.* reason about liveness at the protocol level. This is different from out setting, since their approach is not compositional and they do not directly verify the code.

There is some high-level similarity between interaction permissions in our setting and *escrows* in the GPS program logic [21] for weak-memory programs introduced by Turon *et al.*. Escrows are used as a way to transfer exclusive permission between entities indirectly. This is precisely what interactions permissions are used for as well, since we use them to transfer send event permissions between entities indirectly. One difference is that escrows are assertions that are duplicable (they are not permissions), while interaction permissions cannot be duplicated. Furthermore, there is no analogous notion of progression of an escrow.

7 Conclusion

We have extended the actor services logic [20] developed by Summers and Müller with a notion of sessions. This enables the modular verification of response properties in actor programs where actors change their behaviour according to a protocol. Our technique is particularly effective in topologies such as the ones described in Section 4.12. Using a notion of *session ownership* we can ensure that messages sent to actors will be accepted and do not lead to failures. A fundamental challenge is writing modular specifications to enable the transfer of such ownership to the right actors. We tackle this challenge using *interaction permissions*, which specify at what points in an interaction one can get hold of ownership of which parts in a session. Furthermore, we incorporated initial support for modular fork-join reasoning in actors as part of our session extension.

In the original actor services logic it is sometimes necessary to reason about actors using immutability to achieve modularity. If one enforces such immutability, then one cannot deallocate the affected heap locations and cannot reuse the actors holding the heap locations. Our extension of the logic lifts this limitation for actor programs such as the ones described in Section 4.12.

As we discussed in Section 4.11, our approach cannot deal yet with certain topologies. However, in the same section we provide an argument that our approach can be potentially extended in a natural way to deal with such topologies.

7.1 Future work

Formalization. The current approach must still be formalized and proved sound. We introduced the rules to progress interaction permissions informally and gave an argument why this progression is sound. However, one needs to define a formal model for interaction permissions describing its precise semantics to formally reason about soundness.

The initial extension to sessions in Section 4 only makes small changes to the derivation of non-local actor services. The main change is with respect to how local actor services are proved. Therefore a soundness proof for this initial part should be able to reuse large parts of the original soundness proof for the derivation of non-local actor services.

However, the extension described in Section 5 makes fundamental changes to the actor services structure and the resulting semantics. A soundness proof for this extension might require non-trivial extensions of the original soundness proof for the derivation of non-local actor services.

Branching and recursion. Currently, interaction permissions only support basic interactions. A natural extension would be to support branching. This could be achieved by adding *internal and external choice* to interaction permissions (similar to session types). For example, if interaction permission $(\textsc{send}(E_1) \,.\, R_1) \,\&(\textsc{send}(E_2) \,.\, R_2)$ is held (internal choice), then one could choose to progress $(\textsc{send}(E_1) \,.\, R_1)$ (and hence obtain permission $\textsc{send}(E_1)$) or $(\textsc{send}(E_2) \,.\, R_2)$ (and hence obtain permission $\textsc{send}(E_2)$).

Dually, if interaction permission $(\textsc{rcv}(E_1) \,.\, R_1) \oplus (\textsc{rcv}(E_2) \,.\, R_2)$ is held (external choice), then if $\textsc{rcv}(E_1)$ holds, one could progress $(\textsc{rcv}(E_1) \,.\, R_1)$ and otherwise if $\textsc{rcv}(E_2)$ holds, one could progress $(\textsc{rcv}(E_2) \,.\, R_2)$. One would have to make sure that the companion permissions are always in-sync by enforcing conditions on the session events used.

Another extension would be to permit recursive behaviour. Such an extension is more challenging, because one would need a way to distinguish in which iteration of the recursion one is, to be able to generalize the protocol invariant accordingly.

Dealing with more general topologies. In Section 4.11 we showed that it might be possible to extend the rules to progress interaction permissions to deal with more general topologies. For such an extension having a formal model for interaction permissions is even more important, due to the complexity of the reasoning involved. In Section 4.11 we also hint at other possible extensions which are required to reason about response properties in such topologies.

Session initialization messages. In the current approach one requires partial session ownership whenever one wants to send a message associated with any kind of session. This is too restrictive for some programs. There are actors which always accept the first message of a session (session initialization message). If at the time of reception the actor's session is running, it chooses to deal with the message later. Once the session has finished, the actor then processes one of the next session initialization message received earlier. This can be achieved using selective receive or related approaches (see Section 2.2.3).

An idea to support this would be to mark such session initialization messages explicitly and to accept these messages always. One might need to extend the ActorPL language to then have a way of deferring actions on messages if the session is running. Furthermore to get liveness guarantees when sending session initialization messages, one would have to show that if such a message is received and the session is not running, then it is guaranteed that the session starts and will eventually finish, at which point the actor will again deal with such messages. One challenge is that two different sessions may send session initialization messages to each other. In such cases one can have deadlocks and hence to guarantee sound reasoning, one must have some kind of well-foundedness argument.

Fork-join extensions. Extending the current fork-join approach to deal with an unbounded number of trigger messages and an unbounded number of response messages is required to deal with practical programs. Making the current rules for the composition of actor services with multiple trigger and response messages less restrictive would be important to capture more properties. Furthermore, generalizing the rules other than composition of actor services in the original logic to the more general case is another possible extension.

Parallel sessions. In Figure 4 the manager only deals with one client at a time. However, it would be possible for the manager to deal with multiple clients at the same time if one uses identifiers explicit in the program to distinguish the different sessions that a manager has with the different clients. It would be interesting to extend the current approach to handle such parallel sessions.

References

[1] Pony programming language. https://www.ponylang.org/. Accessed: 2018-04-14.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM.

[5] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, December 2010.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[8] U. de'Liguoro and L. Padovani. Mailbox types for unordered interactions. *CoRR*, abs/1801.04167, 2018.

[9] S. Fowler. An erlang implementation of multiparty session actors. In *ICE*, 2016.

[10] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[11] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[12] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM.

[13] J. Lange and E. Tuosto. Synthesising choreographies from local session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory*, pages 225–239, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight erlang. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages*, pages 95–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[15] R. Neykova and N. Yoshida. Multiparty session actors. *LMCS*, 13:1–30, 2017.

[16] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer-Verlag.

[17] L. Padovani, V. T. Vasconcelos, and H. T. Vieira. Typing liveness in multiparty communicating systems. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, pages 147–162, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[18] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.

[19] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, pages 104–128, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[20] A. J. Summers and P. Müller. Actor services: Modular verification of message passing programs. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS, pages 699–726. Springer Berlin Heidelberg, 2016.

[21] A. Turon, V. Vafeiadis, and D. Dreyer. Gps: Navigating weak memory with ghosts, protocols, and separation. *SIGPLAN Not.*, 49(10):691–707, October 2014.

[22] D. Wyatt. *Akka Concurrency.* Artima Incorporation, USA, 2013.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Modular Verification of Response Properties in Protocol-Based Actor Programs

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Parthasarathy	Gaurav

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 13.05.2018	Gaurav Parthasarathy

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.