Blame-Correct Support for Receiver Properties in Recursively-Structured Actor Contracts

BRAM VANDENBOGAERDE, Vrije Universiteit Brussel, Belgium
QUENTIN STIÉVENART, Université du Québec à Montréal, Canada
COEN DE ROOVER, Vrije Universiteit Brussel, Belgium

Actor languages model concurrency as processes that communicate through asynchronous message sends. Unfortunately, as the complexity of these systems increases, it becomes more difficult to compose and integrate their components. This is because of assumptions made by components about their communication partners which may not be upheld when they remain implicit. In this paper, we bring design-by-contract programming to actor programs through a contract system that enables expressing constraints on receiver-related properties. Expressing properties about the expected receiver of a message, and about this receiver's communication behavior, requires two novel types of contracts. Through their recursive structure, these contracts can govern entire communication chains. We implement the contract system for an actor extension of Scheme, describe it formally, and show how to assign blame in case of a contract violation. Finally, we prove our contract system and its blame assignment correct by formulating and proving a blame correctness theorem.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Specification languages**; *Constraints*.

Additional Key Words and Phrases: design-by-contract, actors, distributed programming languages

ACM Reference Format:
Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. 2024. Blame-Correct Support for Receiver Properties in Recursively-Structured Actor Contracts. *Proc. ACM Program. Lang.* 8, ICFP, Article 254 (August 2024), 29 pages. https://doi.org/10.1145/3674643

1 Introduction

The actor model [Agha 1986] is a model for concurrent computation where an actor is a memory-isolated process and the only means of communication is through asynchronous message sending. This model lends itself to distributed applications where processes assume the role of nodes and asynchronous messages are sent over a network. Unfortunately, as systems increase in complexity, composing multiple actors becomes more difficult [Leesatapornwongsa et al. 2016]. This is because each actor has a set of implicit assumptions about the messages it receives. For instance, an actor might expect messages to contain values satisfying particular constraints. When those constraints are not met actors could start sending unexpected messages, or operations on unexpected values start to fail, leading to unexpected errors in the distributed application.

Design-by-contract [Meyer 1998] is a programming methodology that aims to make assumptions about software components and their interactions explicit. To this end, design-by-contract advocates annotating the software components of a system (e.g., methods, classes, ...) with *contracts* that specify pre- and post-conditions on the state of the system before and after their usage respectively.

Authors' Contact Information: Bram Vandenbogaerde, Vrije Universiteit Brussel, Belgium; Quentin Stiévenart, Université du Québec à Montréal, Canada; Coen De Roover, Vrije Universiteit Brussel, Belgium.

ACM 2475-1421/2024/8-ART254
https://doi.org/10.1145/3674643

Design-by-contract has been studied extensively for sequential programming languages [Dimoulas et al. 2016; Findler and Felleisen 2002; Strickland et al. 2012]. For distributed systems, a typing discipline for the $\pi-$calculus [Milner 1999] called *session types* [Honda et al. 2008] has emerged. These type systems enable defining *global* types that represent the possible sessions each process in the system can participate in. A global specification describes the sequence of message exchanges that each process in the session is expected to follow. These global specifications are then projected onto local specifications for each process and their associated channels. This enables typechecking the process' code to validate whether it adheres to the global specification. Session types have since their inception enjoyed several extensions ranging from more expressive logics [Bocchi et al. 2010], over support for run-time adaptation [Harvey et al. 2021], to forms of run-time monitoring to satisfy security requirements [Jia et al. 2016].

We argue that software systems can also benefit from specifications that start from a local view rather than a global view. Large systems are built by composing components, which are often not designed with specific compositions in mind. Local specifications may therefore be more suited to express the constraints on and assumptions made by these components.

Most session typing systems are limited to decidable logics for the constraints on their messages. Although more expressive logics have been proposed [Bocchi et al. 2010], these are still more constrained and require properties such as *well-assertness*. Contract systems, in contrast, have the benefit that they are verified during the execution of the system, allowing for more (undecidable) properties to be checked during testing and production. This also facilitates verifying actors that change message processing behavior at run time, and enables checking non-trivial properties such as recursive properties that depend on user input.

Contract systems for distributed actor-based programs have been proposed before [Neykova and Yoshida 2017; Scholliers et al. 2015; Waye et al. 2017]. However, they are mostly focussed on specifying the possible messages actors in the system understand by specifying the interface of their *message handlers*. For example, they include contracts that can state which messages are supported and what payload to expect from each message. Unfortunately, they do not support specifying constraints on the communication behavior of the actor *while* it is processing a message. This is significant since actor systems do not rely on traditional *call-return* semantics but communicate through independent messages, and actor systems could exhibit communication behavior that goes beyond simple *request-reply* patterns. Furthermore, message recipients are often implicitly assumed to be the actor that is protected by some contract (in the case of a request), or the sender of the contracted message (in the case of a reply). This is limiting for actor systems that support sending messages containing references to other actors in the system. The communication patterns enabled by such systems can benefit from contracts about where messages are supposed to go and what the content of those messages is supposed to be.

To address these problems, we propose a contract system that can be used to express constraints on the set of message handlers as well as constraints on the communication behavior of a message handler. In short we make the following contributions:

- **Communication contracts:** we propose a novel theory in the context of the classical actor model for *communication contracts*. In contrast to existing work, these contracts can express constraints on the *communication behavior* of a receiver's message handler, without needing to participate in a particular session. Our contracts are *recursively structured* so that they can express complex message chains spanning multiple actors. Furthermore, the contract system supports arbitrary predicates on actor references as constraints on the receiver of the message.
- **Blame assignment:** we develop a comprehensive blame semantics for our *communication contracts*. The semantics is based on the *indy* semantics proposed by Dimoulas et al. [2011], which

we extend with novel blame assignment semantics for contracts on the communication effects of a message handler.

- **Blame correctness:** To prove the correctness of the blame assignment semantics, we first formulate and then prove a *blame correctness theorem* inspired by the provenance-based correctness of Dimoulas et al. [2011]. To the best of our knowledge, we are the first to formulate such a theorem for contracts on the communication effects for actor systems. We also include an executable semantics in *plt-redex* [Klein et al. 2012]

The remainder of this paper is structured as follows. In Section 2 we motivate the need for our contract system using a simple actor-based program that implements the forward flow reactive design pattern. We proceed in Section 3 by giving some background on higher-order contract languages for sequential programs. Next, in Section 4 we introduce our contract system using a set of examples. In Section 5 and Section 6 we give a formal description of our actor and contract language respectively, which we use in Section 7 to prove blame correctness. We conclude with a brief summary on the related work.

2 Motivating Example

Consider the actor-based program visualised in Figure 1. The program consists of three communicating parties: the client, a router, and a multi-media service the client wants to interact with. To interact with the multi-media service, the client must make contact through the *router*. This is so the router can decide which instance of the service to forward the request to based on arbitrary factors such as load. As the router is not capable of handling all the traffic coming from a number of multi-media service instances, each instance of the service is expected to reply directly to the client, instead of forwarding the response through the router again. This communication pattern is often referred to as the "forward flow pattern" [Kuhn et al. 2017].

The code listing below implements this pattern. The actor language used is an implementation of the classic actor model embedded in Racket (a variant of Scheme). First it defines the three *behaviors* of the parties involved in the actor system. The first behavior (line 1) defines the `client`. When an actor is created based on this behavior, it expects to get a reference to the actor that implements the router service. The client has two message handlers. The first handler, called `main`, is used as the entry point of our example. Its purpose is to send a request to the router. Note that these requests are asynchronous and the handler (line 3) completes immediately after the message has been sent. This handler is triggered by a message send on line 16. The second handler (line 4), called `reply`, is used to handle the response of the multi-media service. Line 5 defines the behavior of the router. When an actor is created using this behavior, a list of multi-media services is expected to be passed as an argument. This list is used by the router to decide which service the request will be forwarded to. Ideally, the router selects a suitable service based on *load* using the `pick-service` function called on line 7.

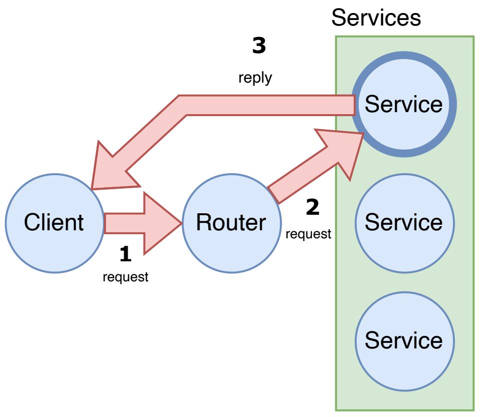

Fig. 1. The "forward flow" pattern. The service selected for processing the request is highlighted with a thick border.

Finally, a behavior for the service is created on line 8. It defines a single handler to handle requests from clients (line 10) . The processing of the request is omitted from this example (line 12). The responsible message handler simply sends an appropriate `reply` back to the original message sender.

```
 1  (define client-behavior
 2     (behavior (router)
 3        (main () (send router request (self)))
 4        (reply (answer) 'omitted)))
 5  (define router-behavior
 6     (behavior (services)
 7        (request (sender) (send (pick-service services) request sender))))
 8  (define service-behavior
 9     (behavior ()
10        (request (sender)
11           'do-work
12           (send sender reply answer-value))))
13  (define service (spawn service-behavior))
14  (define router (spawn router-service (list service)))
15  (define client (spawn client-behavior router))
16  (send client main)
```

The final four lines create actors from these behaviors, and initiate the communication pattern by sending the `main` message to the client behavior. The actors depicted above come with a number of expectations, which can be grouped in three categories. We require the contract system to support their specification and enforcement:

- **Requirement 1: expectations about the interface of an actor:** each actor makes assumptions about the messages that are supported by each of the other actors. For example, the client expects the router to understand the `request` message, while the service expects the client to understand the `reply` message. Furthermore, the content or *payload* of the message is expected to satisfy some constraints. In this example, the constraint is that the first "argument" of the payload is an actor reference that understands the `reply` message. Additionally, there is also an implicit assumption about the type of value sent in the `reply` message.
- **Requirement 2: expectations about the receiver of a message:** The multi-media service is expected to send the reply not to any arbitrary actor, but specifically to the client from which the request originated.
- **Requirement 3: expectations about the communication effects of a message handler:** In this example, the router is expected to forward the client's request to the multi-media service with the lowest load, while the multi-media service is expected to send back the reply directly to the client.

Design-by-contract is a well understood programming methodology where developers annotate program elements (such as functions, methods, ...) with *contracts*. These contracts specify the obligations of the *user* of the program element (also called the client or negative party) and the obligations of the provider of the program element (i.e., the server or positive party). The resulting annotations evaluate to *contract monitors* during program execution, which perform run-time checks to verify that all contracts are satisfied. Blame assignment is an important aspect of design-by-contract. Whenever a contract violation occurs (i.e., one of the stated expectations is not met), a party responsible for the violation must be identified. The responsible party is then assigned *blame* for the contract violation. Blame assignment in the context of design-by-contract for actor systems is non-trivial. Suppose that the router actor advertises itself to other parties using a contract that outlines all expectations (cf. supra). The contract specifies that the router understands a message `request`, and that it will forward such a message to some other (unspecified) actor, and furthermore

```
1  def RouterProtocol() {
2     UsageProtocol: {
3        def start()
4           { (on: request) => forwarded() };
5        def forwarded()
6           { (on: anyMethod) => false };
7     }
8  }
9  def router := object: {
10    def request := provide: request withContract: any -ensures_c(RouterProtocol)->void;
11 } // similar for the service object

12 def ServiceProtocol() {
13    UsageProtocol: {
14       def start()
15          { (on: reply) => replied() };
16       def replied()
17          { (on: anyMethod) => false };
18    }
19 }
```

Listing 1. Workaround for requirement 3 in the system from Scholliers et al.. It defines two protocols, one for each actor (router and service) in the system. Protocols are defined as state machines that start in the `start` method and transition when certain messages are sent. Line 3 defines a transition for the `request` message and transitions the state machine to its final `forwarded` state.

promises that the actor it forwards the message to will send the reply back directly to the client. Now, imagine that the multi-media service does not send the reply back to the client but instead sends it to some other arbitrary actor.

Thus, the contract is violated and blame needs to be assigned. It is unclear which party is to blame. The client is certainly not to blame since it only interacted with the router service and satisfied its contracts. The multi-media service would be the natural party to blame since it did not send the reply back to the client. However, as mentioned before, the router advertises itself with this contract, not the multi-media service. Hence, the *router* is to blame for forwarding the message to an actor that does not process the message in the expected fashion. This brings us to **requirement 4**, which says that blame assignment does not need to be aligned with the boundaries set by actor message handlers.

Existing contract systems [Neykova and Yoshida 2017; Scholliers et al. 2015; Waye et al. 2017] fail to satisfy all four requirements. Whip [Waye et al. 2017] only provides contracts on the interface of an actor, therefore satisfying requirement 1 but not requirements 2 through 4. Thus, contracts in Whip cannot express that the router must forward its request to a downstream service, neither how that service should respond to the forwarded request. Scholliers et al. [2015] propose a *computational contract* system which satisfies requirement 1 and to some extent requirement 3. Their system's support for *requirement 3* is limited to one level of communication only. Expectations about the communication effects generated in response to chains of messages cannot be expressed without causing the contract system to violate requirement 4. A workaround for the lack of support for message chains is shown in Listing 1. For this workaround, two separate contracts have been defined that each monitor the router and the service actor in isolation. In case the service fails to reply directly to the client, the `ServiceProtocol` contract would be violated. The resulting contract system would blame the service actor instead of the router actor for the contract violation, therefore violating requirement 4. Moreover, the contract system leaves receivers unspecified meaning that it does not satisfy requirement 2 as it cannot express that the `request` message should be forwarded to the service.

Multiparty session types [Honda et al. 2008] have been transposed into the contract setting too. Neykova and Yoshida [2017] propose *multiparty session actors* which enable *runtime verification* of their multiparty session types. Their system satisfies requirement 1 and requirement 3, and to a limited extent requirement 2. Neykova and Yoshida do not address blame assignment and therefore do not satisfy requirement 4. Session types express the intended receiver (*req. 2*) of a message through *roles*. These roles, however, are defined when an actor *establishes* a session and cannot change throughout the session's lifetime. This poses two main challenges for expressing that the router should forward the request to a service with the lowest load. First, the list of downstream services can change during the lifetime of the router, which is difficult to capture in a multiparty session contract. Second, the *load* of a downstream service is a dynamic property of the system and cannot be determined using preassigned roles.

Instead, we propose transposing the design-by-contract methodology to the actor setting, and design a Findler-style [Findler and Felleisen 2002] contract system that satisfies all four requirements motivated above:

- *Behavior contracts* specify the expected interface of the actor. Actor languages are higher-order as references to actors can be passed in message payloads to other actors. The contract system itself should support these cases, meaning that behavior contracts should be able to contain other behavior contracts too (*Req. 1*).
- *Receiver contracts* specify the expected receiver of a message and thus accompany the contracts that constrain the tag and payload of a message. These are represented as arbitrary predicates on actor references which enables more expressive dynamic constraints (*Req. 2*).
- *Communication contracts* capture which communication effects are expected to happen. For instance, they can limit the messages an actor is allowed to send while processing a message. Moreover, communication contracts are *dependent* on the message payload of earlier messages, enabling expressing dynamic constraints on communication effects. By exploiting the inherent recursive structure of our contract system, communication contracts can express *communication chains* spanning multiple actors in the system (*Req 3*).
- We develop a *blame assignment* semantics on top of our contract system and prove it correct. Inspired by Findler and Felleisen, our contract system features a standalone *contract monitoring* construct which attaches monitors for checking whether contracts are satisfied to arbitrary values. We propose a system that attaches blame labels to communication contract monitors and sends them along the tag and payload of messages that propagate over the actor system. Thus, our blame assignment semantics correctly blames the router actor of the motivating example by propagating blame labels through message sends (*Req 4*).

In conclusion, we propose a contract system that satisfies all four requirements. Section 4.2 includes a contract in our system for the motivating example. Most contract systems support contracts on the interface of the actor (*Req 1*). Some support contracts on the receiver (*Req. 2*) but only to a limited extent, usually lacking support for expressing dynamic properties. Contracts on communication effects (*Req. 3*) are supported by session types, but cannot be expressed in terms of the payload of earlier messages. Thus, our main contributions are receiver and communication contracts for which we also formulate a correct blame semantics (*Req. 4*). Section 8 features a more detailed comparison to earlier contract and type systems.

3 Background: Sequential Contracts

Design-by-contract [Meyer 1998] is an approach to application design in which the application's components are annotated with contracts, making the expectations towards the client and the supplier of each component explicit. For contracts in higher-order functional programming languages [Findler and Felleisen 2002] these components are *functions*, their *caller* being the client and their *callee* being the supplier. The conditions that should be satisfied by the caller of the function are called *pre-conditions* and are usually defined on the arguments of the function, while the conditions that should be satisfied by the callee are called *post-conditions* and are usually defined on the return value of the function.

```
1  (define/contract (map-positive f lst)
2    (-> (-> (>/c 0) (>/c 0))
3        (listof (>/c 0))
4        (listof (>/c 0)))
5    (if (null? lst)
6        '()
7        (cons (f (car lst)) (map-positive f (cdr lst)))))
```

Listing 2. An example of a function that, given a function that maps positive integers to positive integers (`f`) and a list of positive integers (`lst`) promises to return a list of positive integers. The list is represented as a linked list, with functions `car` and `cdr` taking its head and tail respectively.

We use function `map-positive` depicted in Listing 2 as a running example. The function is defined together with its contract (depicted on lines 2 to 4). All arguments to the `->` function except the last, are contracts on the arguments (domain) of the function. The last argument is a contract on the return value (range) of the function. The contract is a *higher-order* contract, since it contains a contract on a function as one of its argument contracts. The `>/c` contract only matches values that are strictly greater than its argument. Findler and Felleisen [2002] call contracts such as `>/c` *flat*. Flat contracts map values to booleans, where *true* indicates that the contract is satisfied, and *false* that it is violated. In this case the `(>/c 0)` contract maps values to *true* if they are strictly positive integers, and to *false* otherwise.

A call to `map-positive` could look as follows: `(map-positive (lambda (x) (+ x 1)) '(1 2 3))`. For a developer, it is trivial to see that both arguments given to this function satisfy the specified contracts. The contract system too can easily check whether the given list satisfies the specified contract by inspecting the contents of the list when the function is called. This is not the case for the first argument. The reason for this is that the first argument to the `map-positive` function is a function itself, and the contract system cannot predict whether the return value of the function satisfies the specified contract without actually running it. Therefore, checking the contracts on the function is usually delayed until the function is called. To this end, the function is wrapped together with the contract so that the contract on the function can be checked when concrete argument values are known, and when the return value can be computed.

Dependent contracts. Although flat contracts are quite powerful, properties that rely on the input of the function cannot be expressed. For example, using the contracts above, we cannot specify a contract on a function requiring that its output should always be at least twice its input. For this type of contract, a *dependent contract* can be used. Instead of computing the domain and range contracts directly, they are wrapped using a λ-expression that has the arguments of the function as its parameters. For example, to encode the aforementioned contract, one could write: `(->d (>/c 0) (lambda (v) (>/c (* 2 v))))`

Blame assignment. Contracts are different from assertions in that they are able to properly assign *blame* when a violation of one of the specified contracts occurs. For example, when calling the function in Listing 2 as follows: `(map-positive (lambda (x) (+ x 1)) '(-2 1 2))` the contract `(listof (>/c 0))` is violated because the list contains non-positive values.

Clearly, the caller is to blame for supplying the wrong value as an argument. However, when supplying a wrong value for the first argument (e.g., `(lambda (x) (- x 1))`) the blame assignment is less trivial. Whenever the function is called from the body of `map-positive`, `map-positive` is to blame when it supplies a wrong argument to that function, while the caller of `map-positive` is to blame when the `f` function returns the wrong value. Thus blame must be inverted. Finally, a dependent contract might also be to blame when it uses one of the arguments incorrectly. To illustrate, consider the following dependent contract that takes a function as its argument:

```
(->d (-> integer? integer?) (lambda (f) (>/c (f "wrong"))))
```

Here, the function `f` is incorrectly used by the dependent contract. In this case, neither the caller, nor the callee are to blame for the contract violation. Instead, the contract itself is to blame for the contract violation. Correct blame assignment [Dimoulas et al. 2011] therefore considers three involved parties: the caller, the callee, and the contract itself. Correct blame assignment is about assigning blame to the component of the application that controls the values that violate the contract. For first-order values, such as strings and lists, blame is always on the caller of the function when supplying wrong arguments, and on the callee when returning wrong values. For higher-order functions, the caller controls a function passed as a value to the callee. In this case, the caller does not control the arguments of the passed function (the callee does), but it does control its return value. Therefore, the blame labels are swapped for higher-order functions.

4 Communication Contracts in Practice

We introduce our contract language using examples inspired by the *reactive design patterns* [Kuhn et al. 2017]. For each reactive design pattern, we discuss its structure and rationale, and highlight the important aspects that any actor implementing the pattern should satisfy. From this description, we derive an implementation of a software contract in our contract language. Finally, we highlight the novel features of the contract language that facilitate expressing the patterns as contracts. We conclude this section with an overview of our contract language by presenting a summary of its novel contract types. **All examples are included in our replication package [Vandenbogaerde et al. 2024] and are executable by our Racket implementation.**

4.1 Request-Reply Pattern

The "request-reply" pattern is frequently used in a distributed system. It consists of two components: a client and a server. The former sends a request to the latter, while the latter is expected to send the reply. To determine where the reply must be sent to, the client sends a self-reference, called the *reply-to address* as part of the request payload.

To express this pattern as a contract, we use a *message contract*. A *message contract* specifies the expected tag, payload and receiver of a message. The message contract also includes a contract on the receiver of the message (the *receiver contract*) and on its communication effects (the *communication contract*) during message handling. In the code listing below we define a function `request-reply/c` that returns a message contract. The message contract constrains the server to understand a message with tag `request-tag` (line 2) and specifies that the server can expect to receive a payload consisting of an actor reference and any other user-specified contracts (from parameter `request-contracts` on line 1). The message contract also specifies the expected receiver through a contract on the receiver (line 4), but decides to leave the receiver unconstrained through an `any-recipient` contract.

```
1  (define (request-reply/c request-tag reply-tag request-contracts reply-contract)
2    (message/c request-tag                          ;; expected tag
3               (cons actor? request-contracts)      ;; expected payload arguments
4               any-recipient                        ;; expected recipient
5               (lambda (payload)                    ;; expected communication behavior
6                  (ensures/c (list                  ;; of the recipient
7                      (message/c reply-tag
8                                 (list reply-contract)
9                                 (specific-recipient/c (first payload)) ;; expected receiver
10                                unconstrained/c))))))
```

A communication contract (line 6) specifies what messages the server is supposed to send as a result of receiving a message with tag `request-tag`. It specifies that the server must *ensure* that a reply is sent with the expected tag and payload satisfying the expected `reply-contract`. Importantly, it specifies through a message contract `specific-recipient` (line 9) that the reply must be sent to the actor that was passed as part of the payload of the original message. The communication effects on the receiver of the reply (i.e., the client) are not constrained. This is expressed as the `unconstrained/c` contract.

Receiver contracts. The recipient of a message forms an essential part of a distributed system. For example, a reply should end up at the actor that that has sent the request. In contrast to the state of the art, our contract language supports specifying such constraints through *receiver contracts*, which determine contract satisfaction using boolean predicates. Using these boolean predicates within receiver contracts facilitates further extensions without impacting the blame correctness of the contract system. (*Req. 2*).

The `request-reply/c` message contract can be included in a *behavior contract.* A behavior contract is a collection of message contracts that express the shape of messages an actor monitored by the contract should understand[1]. The code listing below illustrates the usage of the previously-defined `request-reply/c` in a behavior contract. The contract expresses that the actor being monitored should understand the `add` message. It also states that the actor can assume that the payload of the message consists of two numbers, in addition to the actor reference specified in the `request-reply/c` function. Furthermore, the actor is expected to reply with a message `result` that includes a number (i.e., the result of adding two numbers together) as its payload.

```
(behavior/c
   (request-reply/c 'add 'result (list number? number?) number?))
```

Behavior contracts are a collection of message contracts and express the interface of an actor. They enable expressing the set of supported messages, and support correct blame assignment in case the message is not understood or a client sends a message that the actor does not support. (*Req. 1*)

4.2 Forward Flow Pattern

In this section we discuss the contract for our motivating example. Message flow between client and server is slowed down when introducing components between a client and a server (e.g., a circuit breaker, rate limiter, load balancer, . . .). These intermediary components are used with a particular purpose for messages flowing from the client to the server, but typically do not add any other interesting behavior to messages flowing in the opposite direction. Therefore, to improve the efficiency of the application, replies originating from the server should be sent back directly to the client instead of flowing back through intermediary components.

In this context, the forward flow pattern [Kuhn et al. 2017] defines three components:

- **A client** that wants to interact with the server, but cannot do so directly because of some intermediary component. It does not know the address of the server and only requests information from the intermediary component.
- **A router** that serves as the intermediary component. On an abstract level, the router receives requests from the client and forwards them to the server after processing them in some way. The actual purpose of this component is irrelevant for this example.

[1]Note that these predicates on payload and receiver go beyond type-related predicates

- **A server** that accepts (potentially modified) requests from the router, processes them, and sends a reply back directly to the client.

To enforce the pattern we create a message contract, depicted below, on a message handler in the router. The contract specifies that the router is expected to receive some message with tag `request-tag` and whose payload satisfies the `request-contracts` (line 2). Similar to the "request-reply" pattern, the client is supposed to add a reference to itself as part of the payload so that it can be answered. The contract continues analogously to the "request-reply" pattern (lines 2-5), however, a reference to the `payload` is kept in scope of the dependent communication contract that monitors the server (lines 10-13). This reference is used to enforce that the reply of the server is sent directly to the original `reply-to` reference (i.e., the client) instead of back to the router.

```
 1  (define (forward-flow/c request-tag request-contracts reply-tag reply-contracts server)
 2    (message/c request-tag
 3               (cons actor? request-contracts)
 4               any-recipient
 5               (lambda (payload)
 6                 (ensures/c  ;; router must forward request to server
 7                   (list (message/c request-tag
 8                                    (cons actor? request-contracts)
 9                                    (specific-recipient server)
10                                    (ensures/c    ;; server must send reply to client
11                                      (message/c reply-tag reply-contracts
12                                                 (specific-recipient (first payload))
13                                                 unconstrained/c))))))))
```

Communication contract chaining. In a distributed system, interactions usually require communication with more than one component. This communication is often *sequential* in nature. One actor might send a message to another, which establishes communication with one or more actors, and so on, before arriving at a final answer for the original request. The nested nature of message contracts enables specifying *message chains* that need to be followed for the contract to be satisfied. (*Req. 3*)

4.3 Correlation Identifier Pattern

The "request-reply" pattern as explained above precludes a client from participating in other simultaneous conversations. This is because the client might send out multiple requests before a single reply arrives, and replies have to be correlated with their original request. This is usually solved by attaching an identifier (commonly referred to as the *correlation identifier*) to the original request which is expected to be included in the corresponding reply. A contract to enforce this pattern at the server side can be constructed as a communication contract and is depicted below:

```
 1  (define (correlation-request-reply/c request-tag reply-tag request-contracts reply-contract)
 2    (message/c request-tag (append (list actor? any?) request-contracts)
 3       any-recipient
 4       (lambda (payload) ;, contract dependent on payload
 5         (define reply-to (first payload))
 6         (define correlation-id (second payload))
 7         (ensures/c (list (message/c reply-tag
 8                                     (list (same-as? correlation-id) reply-contract)
 9                                     (specific-recipient reply-to)
10                                     unconstrained/c))))))
```

The contract starts out like the "request-reply" contract, but differs in its payload contracts. The contracts on the payload (line 2) do not only expect an actor reference for the reply-to address, but also expect a correlation identifier. As the client can choose this identifier, the contract `any?` is used which is satisfied by any value. The communication contract is made *dependent* (starting on line 4) and uses the second value of the payload to specify that the payload of the reply must contain the correlation identifier (`same-as?` on line 8) from the request and a reply value that satisfies the user-specified contract `reply-contract`.

Dependent communication contracts. Contracts on the payload of a message often *depend* on messages received earlier. Our message contracts specify communication contracts that are *dependent*: they accept closures that return a communication contract when applied to the payload of the message. This enables constructing communication contracts based on the payload of any previous message. (*Req 3*).

4.4 Blame Assignment

We illustrate our blame assignment strategy (*Req. 4*) by introducing contract violations in an implementation of the `request-reply` pattern. We introduce the contract violations on two locations: (1) the client sends a payload that does not satisfy the contract, and (2) the server does not send the reply. We run the program twice: the first time we inject only contract violation (1), and the second time we inject contract violation (2). The expected contract and fault-injected program is depicted below:

```
1  (define double/c
2     (behavior/c '() (list (request-reply/c 'double 'answer (list number?) number?))))
3  ;; buggy server that does not satisfy the contract (2)
4  (define double (behavior () (double (reply-to n) (become double))))
5  ;; client that sends the wrong payload (1)
6  (define double-actor (spawn/c double/c double))
7  (send double-actor 'double (self) "wrong")
```

In the first scenario, the client is to blame for supplying the wrong message payload. The blame error below also clearly shows which contracts were tried, but were ultimately not satisfied:

```
contract violation: no matching handler for message "double" found
the following contracts were tried, but did not match the received message:
   error: for tag "double", value "wrong" violated contract #<procedure:number?> ; blaming (line 7, column 0)
blaming (line 7, column 0)
```

In the second scenario, the server is assigned blame since it did not send a reply back to the client. More specifically, the server is assigned blame at the end of its message handler for `double`.

```
contract violation: handler did not send all messages, the following contracts are not satisfied:
  error: message with tag "answer" was not sent ; blaming (line 6, column 21)
```

4.5 Overview of the Contract Language

We conclude this section with an overview of our contract language for constraining actors and their interactions in a distributed system. We start from the *sequential* contract language proposed by Findler and Felleisen, and add new contract types for expressing constraints in an actor system.

Sequential contract. Findler and Felleisen's sequential contract language consists of two contract types: *flat* contracts and *dependent higher-order* contracts. The former type is used for constraints that can be expressed as simple boolean predicates on values. The latter type is used to constrain the interactions between *functions* by attaching contracts to its domain and co-domain.

Behavior contracts. Contracts on actor references require new extensions to the contract system of Findler and Felleisen. Behavior contracts extend the notion of higher-order contracts to *actor references*. They express the expected *message handlers* using a set of *message contracts*.

Value-based contracts. Behavior contracts and sequential contracts form a category of *value-based* contracts, since they both govern interactions with particular values in the language (e.g., numbers, functions or actor references).

Receiver contract. A receiver contract expresses constraints on the expected receiver of a message. It is not a value-based contract since it only operates on message recipients, and is checked when sending/receiving messages.

Message contracts. Message contracts express constraints on the form and shape of messages. They consist of four parts: the expected tag of the message, a value-based contract on the payload of the message, a receiver contract and a communication contract.

Communication contracts. Communication contracts constrain the communication effects of a message handler. In this paper, we present two instantiations of such communication contracts: `ensures/c` and `only/c` which respectively ensure that a particular set of messages is sent, and restrict the set of allowed message sends. Communication contracts consist of a set of message contracts, therefore forming a mutual recursive structure with message contracts. This enables expressing *communication chains* spanning multiple actors in the system.

5 Actor Language

In this section we define an actor language called λ_α. This actor language forms the basis for the discussion about the semantics of our contract language. Its semantics is split in two parts: actor-local semantics and actor-global semantics. The former consists of rules that do not require interaction with other actors in the actor system, while the latter does require interaction. Figure 2 depicts the syntax and actor-local semantics of λ_α.

Actor system and configurations. An actor system $\mathcal{A}$ is a set of configurations C. A configuration represents an active actor in the actor system. An active actor is represented as a tuple $\langle \pi, e, t, M \rangle$ with π being the identifier of the current actor, e the current program of the actor, t a message *trace* and, M the actor mailbox. Message traces are needed for communication contract monitoring, which we discuss in Section 6.4.2. The mailbox is either empty (denoted as $\emptyset$) or has at least one message (denoted by $\langle \tau, v \rangle \cdot M$). A message consists of a message tag τ and a payload v.

Programs. An actor runs a *program*. Our semantics assumes that there is at least one actor in the system that, as its program, defines and creates the other actors in the system. The syntax of a program follows the standard λ-calculus (denoted by terms $\hat{e}$) extended with actor-specific constructs (denoted by $\dot{e}$): send, spawn, self, behavior, and become. The spawn construct takes an expression that evaluates to a behavior and creates a new actor with that behavior. A behavior consists of a set of message handlers each identifiable by a tag. The send constructor enables sending messages between actors in the actor system. As its first argument, it needs an expression that evaluates to an actor reference π. This actor reference will be used as the *receiver* of a message. The second argument is the message tag τ. For simplicity, message tags are not part of the set of values v in λ_α. The third and final argument is the message payload. A self expression can be used by an actor to obtain a reference to itself. Finally, actors can change their behavior using the become construct, after which the actor waits for the arrival of messages in its mailbox.

Program semantics. The λ-calculus subset of our language follows the standard λ-calculus semantics. This semantics is represented as the transition relation $\longrightarrow$ in Fig. 2. Evaluation contexts E are also presented and capture left-to-right evaluation.

Actor-local semantics. Next, we define our actor stepping relation $\longrightarrow_\alpha$. Figure 2 depicts the actor-local semantics. First, a congruence rule [Congr] is defined, which states that when an expression e can be reduced to an expression e', it can also be reduced in the context of a running actor and

$$
\begin{array}{l}
\mathcal{A} \equiv \mathcal{P}(C) \quad M ::= \langle \tau, v\rangle \cdot M \mid \emptyset \\
C ::= \langle \pi, e, t, M\rangle \mid \langle e \rangle \\
e ::= v \mid \hat{e} \mid \dot{e} \qquad \hat{e} ::= e(e) \\
\dot{e} ::= \mathsf{spawn}\ e \mid \mathsf{self} \mid \mathsf{send}\ e\ \tau\ e \\
\qquad \mid \mathsf{become}\ e \mid \mathsf{wait}\ e \\
v ::= \lambda x.e \mid \mathsf{behavior}\ (\tau(x).e \ldots) \mid \pi \\
t \in \mathit{Trace} \qquad \pi \in \mathit{ActorRef}
\end{array}
$$

$$
\begin{array}{l}
E ::= E(e) \mid v(E) \mid \mathsf{spawn}\ E \mid \mathsf{send}\ E\ \tau\ e \\
\qquad \mid \mathsf{send}\ v\ \tau\ E \mid \mathsf{become}\ E \mid \mathsf{wait}\ E \mid \square \\
[\textsc{App}]\ (\lambda x.e_1)v \longrightarrow [x \mapsto v]e_1 \\
[\textsc{Congr}]\ e \longrightarrow e' \Rightarrow \langle \pi, E[e], t, M\rangle \longrightarrow_\alpha \langle \pi, E[e'], t, M\rangle \\
[\textsc{Self-Send}]\ \langle \pi, E[\mathsf{send\ self}\ \tau\ v], t, M\rangle \\
\qquad \longrightarrow_\alpha \langle \pi, E[\mathsf{nil}], t, \langle \tau, v\rangle \cdot M\rangle \\
[\textsc{Become}]\ \langle \pi, \mathsf{become}\ v, t, M\rangle \longrightarrow_\alpha \langle \pi, \mathsf{wait}\ v, t, M\rangle \\
[\textsc{Receive}]\ \langle \pi, \mathsf{wait}\ (\mathsf{behavior}\ (\ldots \tau(x).e \ldots), t, \langle \tau, v\rangle \cdot M\rangle \\
\qquad \qquad \longrightarrow_\alpha \langle \pi, [x \mapsto v]e, t, M\rangle
\end{array}
$$

Fig. 2. Syntax and actor-local semantics of λ_α. Expressions included in the category $\hat{e}$ denote λ-terms while expressions in $\dot{e}$ denote the terms of the actor language.

in an arbitrary evaluation context E. Second is the [Self-Send] rule which adds a message to the actor's mailbox when the recipient of a message coincides with the sender. Third is the semantics (rule [Become]) of the become construct, which is reduced to a wait expression. Finally, [Receive] defines that messages are received when the current behavior matches the first message in the mailbox.

Actor-global semantics. Reductions involving multiple actors are governed by the actor-global semantics. It is defined by a reduction relation $\boldsymbol{\rightarrow}_{\mathcal{A}}$, depicted in Fig. 3, which ranges over actor systems $\mathcal{A}$ instead of individual actor configurations C. An actor system $\mathcal{A}$ is defined as a set of actor configurations. Three additional rules are defined for this reduction relation. First, an additional congruence rule [Congr'] is defined which non-deterministically selects an actor to make a step using the $\longrightarrow_\alpha$ relation. We use the infix operator $\uplus$ to denote a disjoint union. Second, we define a rule [Send] which enables sending messages to other active actors in the system. To send a message, the receiving actor is retrieved from the actor system and the message added to its mailbox. Note that the receiving actor may be in any program state e and is not required to listen explicitly for new messages when the message is sent. Sending a message to a non-existent actor is not defined. Finally, rule [Spawn] is defined which creates a new actor π' with the behavior v.

$$
\begin{array}{l}
[\textsc{Congr'}]\ c \in \mathcal{A}, c \longrightarrow_\alpha c' \Rightarrow \{c\} \uplus \mathcal{A} \boldsymbol{\rightarrow}_{\mathcal{A}} \{c'\} \uplus \mathcal{A} \\
[\textsc{Send}]\ \{\langle \pi_1, E[\mathsf{send}\ \pi_2\ \tau\ v], t_1, M_1\rangle, \langle \pi_2, e, t_2, M_2\rangle\} \uplus \mathcal{A} \\
\quad \boldsymbol{\rightarrow}_{\mathcal{A}} \{\langle \pi_1, E[\mathit{nil}], t_1, M_1\rangle, \langle \pi_2, e, t_2, \langle \tau, v\rangle \cdot M_2\rangle\} \uplus \mathcal{A} \\
[\textsc{Spawn}]\ \{\langle \pi, E[\mathsf{spawn}\ v], t, M\rangle\} \uplus \mathcal{A} \boldsymbol{\rightarrow}_{\mathcal{A}} \{\pi, E[\mathit{nil}], t, M\rangle, \langle \pi', \mathsf{wait}\ v, \emptyset, \emptyset\rangle\} \uplus \mathcal{A} \\
\ \textbf{where}\ \pi' \in \mathit{ActorRef}\ \text{is fresh}
\end{array}
$$

Fig. 3. Actor-global semantics

6 Contract Language

This section introduces the formal syntax and semantics of our contract language. It is structured as follows. First, we describe the syntax of our contract language. Next, we discuss the semantics of our novel *receiver contracts* which express constraints on the receiver of the message. Finally, we conclude with communication contract monitoring semantics.

6.1 Syntax

The extensions to the syntax of λ_α are depicted in Fig. 4. In the remainder of this paper, we call the resulting language $\lambda_{\alpha/c}$. First, we extend the expression syntax e with a *contract monitoring* construct $mon_l^{j,k}\ \kappa\ e$. This construct serves as a barrier between the party j supplying expression e to the client k. Subscript l denotes the party that supplies the contract to the monitoring expression. This barrier ensures that any value flowing between party k and j is governed by contract κ. Next, we define three additional values: $\kappa, \kappa r$ and κc. These values represent the category of value-based contracts, receiver contracts, and communication contracts respectively.

Value-based contracts. Contract types in the category of value-based contracts can be directly used in the monitoring construct (cf. supra) to govern interactions with the entity given by expression e. For the sequential subset of the λ_α language, borrowing terminology from CPCF, we define contract types for dependent higher-order function contracts ($\kappa_1 \rightarrow \lambda x.e$) and flat contracts ($\mathsf{flat}(e)$) in a similar way as CPCF [Findler and Felleisen 2002]. Flat contracts wrap regular functions that are expected to behave as boolean predicates for the property that they want to check. Thus, the boolean predicate e is expected to return *true* in case the contract is satisfied and *false* otherwise. Dependent higher-order contracts are used for monitoring interactions with a function by checking the domain contract κ_1 on the argument of the function, and the range contract obtained from applying function $\lambda x.e$ using monitored argument on the return value of the function. Interactions with actor references are governed by a *behavior* contract. These are denoted by $\mathsf{behavior/c}$ in our syntax. The behavior contract expects a set of message contracts (similar to *union contract* [Freund et al. 2021]) that specify which messages must be understood by the monitored actor reference.

Message contracts. Message contracts express constraints on the content of a message, as well as the supposed receiver and the communication effects of that receiver. A message contract κm consists of four parts. First, it specifies the expected message tag τ, which should correspond exactly to the message being sent/received. Second is a dependent contract that should reduce to a contract on the receiver of the message when supplied with the payload of the message. The third argument is the contract on the payload of the message, which should be a *valued-based contract* as denoted by κ. Finally, the fourth argument is a dependent contract again, which should reduce to a contract on the communication effects *of the message's receiver* when supplied with the payload of the message. Alternatively, message contracts can also be *empty*, denoted by $\emptyset$, which signifies that there are no constraints on the message.

Receiver contracts. Receiver contracts express constraints on the receiver of a message. They function similarly to flat contracts, but cannot be used in the position of a value-based contract. The expression e in $\mathsf{receiver}(e)$ is supposed to return a boolean predicate, that returns *true* when the given receiver is allowed to receive the message and *false* otherwise. We omit additional predicates from the formal syntax of our language, such as `actor-eq?`, which checks whether two actor references are the same. However, those predicates are needed in a practical implementation of our contract system in order to, for instance, implement contracts that only allow a specific set of receivers.

Communication contracts. Contracts on the communication effects of an actor are described by the κc category. They consist of message contracts κm, *ensures* contracts $\mathsf{ensures/c}$ and *only* contracts. We describe their semantics extensively in the Section 6.3. Note that the category of values v also includes the contract monitoring construct. This enables contract monitors to be passed as values, which is needed for passing monitored actor references. We explain this addition in more detail in Section 6.3.

$$
\begin{aligned}
e &::= \ldots \mid \mathit{mon}_l^{j,k}\ e\ e \qquad v ::= \ldots \mid \kappa \mid \kappa r \mid \kappa c \mid \mathit{mon}_l^{j,k}\ v\ v \\
\kappa &::= \kappa \rightarrow \lambda x.e \mid \mathsf{flat}(e) \mid \mathsf{behavior/}c\ (\kappa m \vee \ldots \vee \kappa m) \\
\kappa m &::= \mathsf{message/}c\ \tau\ (\lambda x.e)\ \kappa\ (\lambda x.e) \mid \emptyset \qquad \kappa r ::= \mathsf{receiver}\ (e) \\
\kappa c &::= \kappa m \mid \mathsf{only/c}\ (\kappa m, \ldots) \mid \mathsf{ensures/c}\ (\kappa m, \ldots) \qquad j, k, l \in \mathit{BlameLabel}
\end{aligned}
$$

Fig. 4. Syntax of $\lambda_{\alpha/c}$, extensions to syntactical categories of $\lambda_\alpha$are denoted by "..."

6.2 Semantics for Sequential Contracts

We first extend the evaluation contexts with contexts for our new monitoring construct (depicted in Fig. 5). For the sequential subset of $\lambda_{\alpha/c}$, rules [MonFlat] and [MonFun] define the reduction semantics for monitoring flat and higher-order dependent contracts. A monitor on a flat contract is reduced to the check^j expression by applying the boolean predicate $\lambda x.e$ on the value v_2. The check^j expression takes a blame label j, the boolean v_1 resulting from the function application, and the "original" value v_2, and returns v_2 if v_1 is *true*. Blame labels keep track of the party that should be blamed for a contract violation. Whenever v_1 is *false* a *blame error* is generated, assigning blame to party j. A monitoring rule for higher-order contracts is depicted in rule [MonFun]. A monitor on a higher-order contract is reduced to the following λ−expression:

$$\lambda x_2.\mathit{mon}_l^{j,k}([x_1 \mapsto \mathsf{mon}_l^{k,l}\ \kappa_1\ x_2]\ \kappa_2)\ (v\ (\mathsf{mon}_l^{k,j}\ \kappa_1\ x_2))$$

This expression serves as a wrapper for the monitored function v. Contract monitoring proceeds as follows when the λ−expression is applied with an argument for x_2. First, the function argument is replaced with a value monitored by contract κ_1. Then, the return value of the function is monitored by the range contract κ_2. To obtain this range contract, the dependent contract $\lambda x_2.\kappa_2$ is applied to the argument of the function. Importantly, the argument of the function (as captured by x_2) is monitored by the domain contract κ_1 before it is passed to the code that returns the range contract κ_2. The reason for this is that the range contract has to be able to assume that the domain contract on the argument of the function holds.

$$
\begin{aligned}
& e ::= \ldots \mid \mathsf{check}^j\ e \qquad E ::= \ldots \mid \mathsf{check}^j\ E\ \mid \mathsf{mon}_l^{j,k}\ E\ e \mid \mathsf{mon}_l^{j,k}\ v\ E \\
& [\textsc{MonFlat}] \quad E[\mathsf{mon}_l^{j,k}\ \mathsf{flat}(\lambda x.e)\ v_2] \rightarrow E[\mathsf{check}^j\ ([x \mapsto v_2]\ e)\ v_2] \\
& [\textsc{MonFun}] \quad E[\mathsf{mon}_l^{j,k}\ (\kappa_1 \rightarrow \lambda x_1.\kappa_2)\ v] \rightarrow \\
& \qquad\qquad \lambda x_2.\mathit{mon}_l^{j,k}([x_1 \mapsto \mathsf{mon}_l^{k,l}\ \kappa_1\ x_2]\ \kappa_2)\ (v\ (\mathsf{mon}_l^{k,j}\ \kappa_1\ x_2))
\end{aligned}
$$

Fig. 5. Contract monitoring semantics for the sequential subset of $\lambda_{\alpha/c}$

Rules [MonFlat] and [MonFun] also include the formal semantics for blame assignment. It is identical to the so-called *indy* semantics [Dimoulas et al. 2011, 2012]. The blame labels are swapped when checking the domain contract against the argument of the function. This ensures that the client is blamed when the domain contract is not satisfied. Moreover, it ensures correctness of blame assignment for higher-order functions. The contract itself is also modeled as its own party, called l, so that when the code in the contract reduces to a contract violation, the contract itself is blamed.

6.3 Sender-Side Contract Monitoring

6.3.1 Receiver Contracts. Recall that message contracts include a contract on the receiver of a message, restricting which actors can receive the messages specified by the message contract. Receiver contracts can be seen as a variation on *flat contracts*: whenever the expected receiver is given it returns a true value, otherwise false. Thus, both receiver and flat contracts are checked using the check meta-function. Even though receiver contracts can be denoted by arbitrary expressions, we propose two primitive contracts for the sake of discussion: `any-recipient` and `specific-recipient`, their semantics are depicted in Fig. 6. Rule [AnyRecipient] defines the semantics for `any-recipient`. Since any actor reference should satisfy this contract, actor reference α is simply returned from this contract check. Rules [SpecificRecipient1] and [SpecificRecipient2] specify the contract checking rules for `specific-recipient`. The former rule returns the monitored actor reference α whenever $\alpha_1 = \alpha$, while the latter results in a blame error if the actor reference does not correspond to the expected actor reference.

$$\begin{array}{rl}
[\textsc{AnyRecipient}] & \mathsf{check}^{j}\ \texttt{any-recipient}\ \alpha \longrightarrow \alpha \\
[\textsc{SpecificRecipient1}] & \mathsf{check}^{j}\ (\texttt{specific-recipient}\ \alpha_1)\ \alpha_1 \longrightarrow \alpha_1 \\
[\textsc{SpecificRecipient2}] & \mathsf{check}^{j}\ (\texttt{specific-recipient}\ \alpha_1)\ \alpha_2 \longrightarrow \mathsf{blame}^{j} \\
 & \text{given that } \alpha_1 \neq \alpha_2
\end{array}$$

Fig. 6. Receiver contracts. The `any-recipient` contract allows any receiver, while `specific-recipient` only matches a specific recipient.

6.3.2 Stacked Contract Monitors. Contract monitors on actor references can be arbitrarily stacked, adding additional constraints to the actor reference. The code listing below depicts an actor system comprising two actors α_1 and α_2, monitored by contracts κ_1 and κ_2 respectively. The contract κ_2 includes a message contract κm that is structured as follows. The contract expects τ as the tag of the

$$\begin{array}{ll}
\kappa_m = \mathsf{message/c}\ \tau\ \_\ \kappa_3\ \_ & \kappa_2 = \mathsf{behavior/c}\ (\kappa_m) \\
x_1 = \mathsf{mon}_{l}^{j,k}\ \kappa_1\ \alpha_1 & x_2 = \mathsf{mon}_{l'}^{j',k'}\ \kappa_2\ \alpha_2 \\
\mathsf{send}\ x_2\ \tau\ x_1 &
\end{array}$$

message to match the send expression. For the payload the contract expects an actor reference that satisfies a contract κ_3. The other parts of the message contract are unimportant for this discussion.

The message send at the end of the code listing causes α_2 to receive a message with tag τ. As its payload, the message will contain a *monitored* actor reference of α_1. This is because it is monitored by contract κ_1 but also by the contract on the payload specified in κ_m. More specifically, the contract κ_2 on actor α_2 expects an actor reference that behaves according to contract κ_3, but the passed actor reference is exposed under *another* contract κ_1 that must be satisfied too. Essentially, the resulting monitored actor reference has a *stack* of contract monitors, which comes with a stack of *blame labels* since contract monitors can originate from different parties in the source program.

To support these stacks of contract monitors, we introduce three meta-functions into our semantics: stack_p, stack_r and stack_c, which generate stacked versions of value-based contracts on the payload, receiver contracts and communication contracts respectively. As stacked contracts can only monitor actor references (all other contract monitors reduce immediately), we define these meta-functions to take contract monitors on actor references. Their definition is depicted in Fig. 7.

$\boxed{\text{Value-based contracts on the payload}}$

$\text{stack}_p\ (\text{mon}_l^{j,k}\ \kappa_1\ v_1)\ v_2 = (\text{stack}_p\ v_1\ (\text{mon}_l^{k,j} \kappa_p\ v_2))$

$\textbf{where}\ \ \kappa_1 = \text{behavior/c}\ \kappa m_1 \vee \ldots \vee \kappa m_n \qquad \text{message/c}\ \tau\ \_\ \kappa_p\ \_ \in \{\kappa m_1, \ldots, \kappa m_n\}$

$\text{stack}_p\ \pi\ v_2\ =\ v_2$

$\boxed{\text{Receiver contracts}}$

$\text{stack}'_r\ (\text{mon}_l^{j,k}\ \kappa_1\ v_1)\ c = \text{stack}'_r\ v_1\ (\lambda x.\lambda y.(\text{check}^k\ (\kappa r\ y)\ (c\ x\ y)))$

$\text{stack}'_r\ \pi\ c = c\ \pi$

$\textbf{where}\ \kappa_1 = \text{behavior/c}\ \kappa m_1 \vee \ldots \vee \kappa m_n \qquad \text{message/c}\ \tau\ \kappa r\ \_\ \_ \in \{\kappa m_1, \ldots, \kappa m_n\}$

$\text{stack}_r\ v = \text{stack}'_r\ v\ (\lambda x.x)$

$\boxed{\text{Communication contracts}}$

$\text{stack}_c\ (\text{mon}_l^{j,k}\ \kappa_1\ v_1) = (\lambda x.\text{monc}^j\ \kappa c^j\ x) \circ \text{stack}_c\ v_1$

$\textbf{where}\ \ \kappa_1 = \text{behavior/c}\ \kappa m_1 \vee \ldots \vee \kappa m_n \qquad \text{message/c}\ \tau\ \_\ \_\ \kappa c \in \{\kappa m_1, \ldots, \kappa m_n\}$

$\text{stack}_c\ \pi = \emptyset$

Fig. 7. Rules for unstacking contract monitors on actor references and extracting their payload, communication, and receiver contracts.

stack_p is defined by two rules. The first rule extracts the contract monitor from κ_2 on v_1 and looks for a matching message contract. If a matching message contract is found, the contract on the payload κ_2 is checked against the value v_2. The resulting monitored value is passed to the other contracts that might be in value v_1, in order to attach their contracts to the monitored payload value. The second rule returns the payload value if the actor reference is not monitored by any contract. This terminates the recursive process.

Note that the blame labels are swapped while checking κ_2 on v_2, meaning that the negative party (i.e., the client) is to blame for the contract violation. This is consistent with *indy* semantics.

To stack receiver contracts, we proceed in a similar fashion as the contracts on the payload except that the semantics is expressed in a *continuation-passing style.* This is because the actual actor reference to check the contracts against is only known after the bottom of the stack has been reached. From there, the receiver contracts must be added in reverse order to the actor reference by using check expressions. Note that the *negative* party is assigned blame. Indeed, the client is responsible for selecting the message receiver and is thus to blame for an unsatisfied contract.

Finally, stacking communication contracts proceeds in the manner described in the beginning of this section. However, instead of stacking check^j expressions, we stack communication contract monitors which take the form of a λ-expression containing a monc expression. We discuss the reduction of these expressions in Section 6.4.1.

6.3.3 Enhanced Sends. To complete the semantics for contract monitoring at the send site of messages, an intermediary message sending syntax that adds a communication contract in addition to the receiver, tag and payload is required. The inclusion of the communication contract in the message send expression is necessary to send the contract alongside the tag and payload to the receiver so that the receiver can check whether its handler adheres to the communication contract. We call these new expressions *enhanced message sends* and use $\overline{\text{send}}\ e_\kappa\ e\ \tau\ e$ for their notation.

We replace the message sending rules of the actor system introduced in Section 5 with a new rule that reduces a message send expression to an enhanced message send expression. The appropriate contracts are extracted from the contract monitors around the receiver of the message v_1 using meta-functions stack_c, stack_r and stack_p (cf. supra). The altered semantics is depicted in Fig. 8.

$$e ::= \ldots \mid \overline{\mathsf{send}}\ \kappa m\ e\ \tau\ e \mid \mathsf{monitored}\ e \qquad M ::= \ldots \mid \langle \tau, \kappa m, v\rangle \cdot M$$

$$E ::= \ldots \mid \overline{\mathsf{send}}\ \kappa m\ E\ \tau\ e \mid \overline{\mathsf{send}}\ \kappa m\ v\ \tau\ E$$

$$[\textsc{Send}]\ \mathsf{send}\ v_1\ \tau\ v_2 \longrightarrow \overline{\mathsf{send}}\ (\mathsf{stack}_c\ v_1)\ (\mathsf{stack}_r\ v_1)\ \tau\ (\mathsf{stack}_p\ v_1\ v_2)$$

$$[\textsc{E-Send}]\ \{\langle \pi_1, E[\overline{\mathsf{send}}\ v_\kappa\ \pi_2\ \tau\ v], t_1, M_1\rangle, \langle \pi_2, e_2, t_2, M_2\rangle\} \uplus \mathcal{A}$$
$$\longrightarrow_{\mathcal{A}} \{\langle \pi_1, E[nil], t_1, M_1\rangle, \langle \pi_2, e_2, t_2, (\tau, v_\kappa, v)\cdot M_2\rangle\} \uplus \mathcal{A}$$

$$[\textsc{E-SendSelf}]\ \langle \pi_1, E[\overline{\mathsf{send}}\ v_\kappa\ \pi_2\ \tau\ v], t_1, M_1\rangle \longrightarrow_\alpha \langle \pi_1, E[nil], t_1, (\tau, v_\kappa, v)\cdot M_1\rangle$$

Fig. 8. Send-site contract monitoring rules.

As depicted by rule [E-Send], information from an enhanced message send is propagated to the receiving actor by putting an *enhanced message* in its mailbox. This enhanced message is represented by a three tuple (v_κ, τ, v) which includes the communication contract v_κ alongside the tag τ and payload v. Rule [E-SendSelf] is similar, but puts the message in the mailbox of the sending actor. Enhanced messages are handled differently from regular messages. This is because the contract system has to ensure that the communication effects of the receiving actor satisfy the communication contracts. We introduce their semantics in the next section.

6.4 Receive-Side Contracts

6.4.1 Communication Contract Monitors. For monitoring a communication contract, we introduce a communication contract monitoring expression monc. The purpose of this expression is to check whether outgoing messages satisfy the communication contract κc. A message consists of the four values: a communication contract, the receiver of the message, its tag and its payload. These values are captured in a four-tuple $(\lambda x.\mathsf{monc}\ \kappa c'\ x, \alpha, \tau, v)$, such that the monitor expression becomes $\mathsf{monc}\ \kappa c(\lambda x.\mathsf{monc}\ \kappa c\ x, \alpha, \tau, v)$. The second communication contract originates from the contract monitor in a send expression, and has to be combined with the communication contract of monc. monc expressions are created by the stack_c meta-function which introduces them as a λ-expression $\lambda x.\mathsf{monc}\ \kappa c\ x$, where x is expected to be the aforementioned four-tuple.

Figure 9 depicts the monitoring semantics for checking communication contracts on messages. The result of checking a communication contract on a message represented by a four-tuple, is a *monitored* four-tuple where the appropriate contract is checked on each of its constituents. The first three rules depicted represent contract checking for message/c, only/c and ensures/c respectively. Note that a new communication contract that governs the communication effects of the receiver of the message can originate from both a monitored actor reference and from the monitored context of the sender of the message. We represent this combination as a function composition $e_\kappa \circ e'_\kappa$. This is possible since a communication contract monitor is always expected to be wrapped into a λ-expression. This composition is defined as a normal function composition.

The first rule depicts monitoring rules for message/c contracts. To check this contract, the message tag should match the message tag in the message contract, the receiver should match the receiver contract $\lambda x_1.e_1$, and the payload should match the contract on the payload κ. A message contract also contains a communication contract $\lambda x_2.e_2$ which governs the communication effects

$$\boxed{\text{Message contract}}$$

$$\mathsf{monc}^j\ (\texttt{message/c}\ \tau\ (\lambda x_1.e_1)\ \kappa\ (\lambda x_2.e_2))^j\ (e'_\kappa, v_1, \tau, v_2)$$
$$= (e_\kappa \circ e'_\kappa, ((\mathsf{check}^j([x_1 \mapsto v_2]\ e_1)\ v_1), \tau, \mathsf{mon}_j^{j,j}\ \kappa\ v_2)$$
$$\textbf{where } e_\kappa = \lambda x.\mathsf{monc}^j\ ([x_2 \mapsto v_2]e_2)\ x$$
$$\mathsf{monc}^j\ (\texttt{message/c}\ \tau\ \_\ \_\ \_)\ \_ = \mathsf{blame}^j$$

$$\boxed{\text{Only contract}}$$

$$\mathsf{monc}^j\ (\texttt{only/c}\ (\ldots, \texttt{message/c}\ \tau\ (\lambda x_1.e_1)\ \kappa\ (\lambda x_2.e_2), \ldots)\ (\kappa c, v_1, \tau, v_2)$$
$$= \mathsf{monc}^j\ (\texttt{message/c}\ \tau\ (\lambda x_1.e_1)\ \kappa\ (\lambda x_2.e_2))\ (\kappa c, v_1, \tau, v_2)$$
$$\mathsf{monc}^j\ \texttt{only/c}\ (\_)\ \_ = \mathsf{blame}^j$$

$$\boxed{\text{Ensures contract}}$$

$$\mathsf{monc}^j\ (\texttt{ensures/c}\ (\ldots, \texttt{message/c}\ \tau\ (\lambda x_1.e_1)\ \kappa\ (\lambda x_2.e_2), \ldots)^j\ (\kappa c, v_1, \tau, v_2)$$
$$= \mathsf{monc}^j\ (\texttt{message/c}\ \tau\ (\lambda x_1.e_1)\ \kappa\ (\lambda x_2.e_2))^j\ (\kappa c, v_1, \tau, v_2)$$
$$\mathsf{monc}^j\ (\texttt{ensures/c}\ (\_)\ (\overline{\mathsf{send}}\ \kappa c\ v_1\ \tau\ v_2)^j = (\emptyset, v_1, \tau, v_2)$$

Fig. 9. Monitoring semantics for communication contracts on messages.

of the receiver of the message. Whenever one of the aforementioned conditions is not satisfied, the contract is violated and a blame error on party j is generated.

Next, we define the semantics of the only/c and ensures/c contract. Their semantics is mostly identical, except for how they handle missing message contracts. For both contracts, whenever a message contract is found that matches the message, that contract is selected and checked recursively. However, in case a message contract matching the message's tag is not found, an ensures/c contract simply leaves the message untouched and returns it as is. The only/c contract considers this case to be a contract violation instead and reduces to a blame error. This is because an only/c contract specifies what messages are *allowed* to be sent. Therefore, a missing match for the message means that it was not allowed to be sent.

6.4.2 Trace Checking. At the end of an actor's turn, the contract system checks whether all the messages specified in the communication contract have been sent. To this end, we overload the monc notation to include monitoring rules over messages traces. In this case monc expects three arguments: the communication contract itself, a message trace, and a value. Whenever the message trace satisfies the contract, the value is returned unmodified. Otherwise, a blame error is returned. For this we consider two cases: the ensures/c contract, and composed communication contracts of the form $\kappa c_1 \circ \kappa c_2$. The cases for only/c and message/c are straightforward as they do not put any constraints on the message traces. Figure 10 depicts the cases mentioned above.

The contract system verifies that all message contracts in an ensures/c contract have a corresponding message in the trace. This is covered by the first case. If the contract is satisfied, the new behavior v is returned from the meta-function. Otherwise, the contract system returns a blame error. Contract composition proceeds as follows: the contract κc_1 is checked first against the message trace, followed by the checking of contract κc_2. A final case expresses recursive blame propagation.

$$\boxed{\text{Ensure contract}}$$

$$\texttt{monc}^j\ (\texttt{ensures/c}\ (\kappa m_1, \ldots, \kappa m_2))^j\ t\ v =$$
$$\begin{cases} v & \text{if } \forall\ (\texttt{message/c}\ \tau\ \_\ \_\ \_) \in \{\kappa m_1, \ldots \kappa m_2\}\ :\ (\tau, \_) \in t \\ \text{blame}^j & \text{otherwise} \end{cases}$$

$$\boxed{\text{Contract composition}}\quad \texttt{monc}^j\ (\kappa c_1 \circ \kappa c_2)\ t\ v = \texttt{monc}\ \kappa c_2\ (\texttt{monc}\ \kappa c_1\ t\ v)\ v$$

$$\boxed{\text{Blame propagation}}\quad \texttt{monc}^j\ \_\ \text{blame}^j\ \_ = \text{blame}^j \qquad \boxed{\text{Other cases}}\quad \texttt{monc}\ \_\ \_\ v = v$$

Fig. 10. Definition of the checktrace function. It takes three arguments: the communication contract, the message trace, and the new behavior of the actor. If the contract is satisfied, the behavior is returned. Otherwise, a blame error is generated.

6.4.3 Monitored Contexts. Our actor system uses *enhanced messages* to denote messages that have to be processed in accordance with a communication contract. Such messages are formally denoted by a triple $(\tau, \kappa c, v)$, where τ and v fulfil their usual roles, and v_κ is used as the communication contract monitor that will govern all the communication while processing the message.

For monitoring the communication effects of the actor system, we introduce the concept of monitored context. The context consists of a contract monitor that governs the communication effects, and an expression which contains the program that is being monitored for its communication effects. Thus we write monitored v_κ e to denote that program e is monitored by the communication contract monitor v_κ. Existing message handlers become monitored when receiving an enhanced message from another actor. The communication contract monitor is then extracted from the enhanced message, and used as the communication contract monitor in the monitored expression. The semantics of programs in monitored contexts remains largely the same for most types of expressions in our language. The reduction rules for monitored contexts are given in Fig. 11.

$$\begin{aligned}
&[\text{E-Receive}]\ \langle \pi, \text{wait (behavior } (\ldots (\tau\ (x)\ e) \ldots)), t, (\tau, e_\kappa, v) \cdot M \rangle \\
&\quad \rightarrow_\alpha \langle \pi, \text{monitored } e_\kappa\ ([x \mapsto v]\ e), t, M \rangle \\
&[\text{M-Congruence}]\ \hat{e} \rightarrow \hat{e}' \Rightarrow \langle \pi, \text{monitored } e_\kappa\ E[\hat{e}], t, M \rangle \rightarrow_\alpha \langle \pi, \text{monitored } e_\kappa\ E[\hat{e}'], t, M \rangle \\
&[\text{M-Spawn}]\ \{\langle \pi, \text{monitored } e_\kappa\ E[\text{spawn } v], t, M \rangle\} \\
&\quad \rightarrow_{\mathcal{A}} \{\langle \pi, \text{monitored } e_\kappa\ E[\pi'], t, M \rangle, \langle \pi', \text{wait } v, \emptyset, \emptyset \rangle\} \\
&[\text{M-Send}]\ \text{send } v_1\ \tau\ v_2 \rightarrow \overline{\text{send}}\ e'_\kappa\ v_3\ \tau\ v_4 \Rightarrow \langle \pi, \text{monitored } e_\kappa\ E[\text{send } v_1\ \tau\ v_2], t, M \rangle \\
&\quad \rightarrow_\alpha \langle \pi, \text{monitored } e_\kappa\ E[\text{dosend } (e_\kappa\ (e'_\kappa, v_3, \tau, v_4))], t, M \rangle \\
&[\text{M-ESend}]\ \{\langle \pi, \text{monitored } e_\kappa\ E[\overline{\text{send}}\ \kappa c'\ \pi'\ \tau\ v_2], t_1, M_1 \rangle, \langle \pi', e, t_2, M_2 \rangle\} \uplus \mathcal{A} \\
&\quad \rightarrow_{\mathcal{A}} \{\ \langle \pi, \text{monitored } e_\kappa\ E[nil], (\tau, v_2) \cdot t_1, M_1 \rangle, \langle \pi', e_2, t_2, (\tau, \kappa c', v) \cdot M \rangle\} \uplus \mathcal{A} \\
&[\text{M-ESendSelf}]\ \langle \pi, \text{monitored } e_\kappa\ E[\overline{\text{send}}\ \kappa c'\ \pi\ \tau\ v_2], t, M \rangle \\
&\quad \rightarrow_\alpha \langle \pi, \text{monitored } e_\kappa\ E[\text{nil}], (\tau, v_2) \cdot t, (\tau, \kappa c', v_2) \cdot M \rangle \\
&[\text{M-Become}]\ \langle \pi, \text{monitored } e_\kappa\ E[\text{become } v]\ , t, M \rangle \rightarrow_\alpha \langle \pi, e_\kappa\ t\ v, \emptyset, M \rangle
\end{aligned}$$

Fig. 11. Adapted rules for monitored contexts. Most expression types behave in the same way, except for send and become which are intercepted and checked against the communication contract.

The only expressions affected by monitored contexts are the send expressions and the become expressions. Message sends are intercepted in the [M-Send] rule. Assuming that a reduction exists in the base semantics that reduces a message send to an enhanced message send, we intercept that message send and apply the communication contract to it. As the communication contract monitor returns a four-tuple when the contract is satisfied, meta-function dosend transforms this tuple into an enhanced message send. We also adapt rules semruleESend and [ESend-Self] into the monitored context as [M-ESend] and [M-ESendSelf]. For these rules, no contract monitoring is applied but the message trace t is updated to include the tag and payload of the send message. This trace is used in rule [M-Become] to check that all messages specified in the contract have been sent during the turn of the actor.

7 Theoretical Properties

Blame assignment is an important component of a contract system. It assigns blame to the responsible party when a contract is violated. However, as blame labels propagate through the program in contract monitors, showing that this blame assignment is *correct* is not self-evident. We use the theoretical results from Dimoulas et al. [2011] as the foundation for our blame correctness proof. Dimoulas et al. track ownership throughout the execution of the program. Informally, their blame correctness theorem states that blame labels should align with ownership which means that party is only blamed if the value violating the contract originated from that party.

The system tracks ownership as values are passed from one component of the application to another. For example, passing an argument to a function owned by another party, causes the ownership of that argument to transfer to the function. The key insight is that we can track these ownership changes independently from contract checking and blame assignment, and prove that the alignment of these two systems entails blame correctness.

7.1 Ownership Annotations

Similar to Dimoulas et al. we introduce an ownership annotation $[\![e]\!]^j$ meaning that expression e is owned by (or originates from) party j. Source programs may include these annotations as long as the resulting program satisfies the well-formedness condition. This well-formedness condition can be derived syntactically from the program's source code. We formalize the well-formedness condition using a judgment $\vdash$. We write $j \vdash e$ to mean that program e is well-formed under owner j.

Definition 7.1. A program e is *well-formed* under owner j iff $j \vdash e$ holds.

We proceed by defining this judgment for the sequential and actor subset of our language. The judgment for the sequential subset of our language is identical to the judgment from Dimoulas et al., and is depicted in Fig. 12. The judgment is mostly structural as most expressions cannot contain ownership annotations at the source level. Ownership annotations can only be introduced at contract monitors (i.e., using *mon* expressions). This is because a contract monitor introduces an *ownership boundary* between a client j and the server k; any value at the server-side is owned by the server and vice versa. The last rule in Fig. 12 defines this property.

Definition 7.2. An actor configuration $\langle \pi, e, t, M \rangle$ is well-formed for an owner j iff $j \vdash e$ holds.

Similarly, we can define well-formedness for the entire actor system $\mathcal{A}$. We say that $\mathcal{A}$ is well-formed if all the actor configurations in $\mathcal{A}$ are well-formed. We write this property as $j \vdash_{\mathcal{A}} \mathcal{A}$ Thus ownership is a local property of the actor. However, ownership can be *transferred* between parties using function application and message sends, which we discuss in the section that follows.

$$\frac{j \vdash e_1 \qquad j \vdash e_2}{j \vdash e_1\ e_2} \qquad \frac{j \vdash e}{j \vdash \mathsf{spawn}\ e} \qquad \frac{j \vdash e_1 \qquad j \vdash e_2 \qquad j \vdash e_3}{j \vdash \mathsf{send}\ e_1\ e_2\ e_3} \qquad \frac{j \vdash e}{j \vdash \mathsf{wait}\ e}$$

$$\frac{j \vdash e}{j \vdash (\lambda x.e)} \qquad \frac{j \vdash e_1 \qquad \dots \qquad j \vdash e_n}{j \vdash \mathsf{behavior}(\tau_1\ x_1.e_1, \dots, \tau_n\ x_n.e_n)} \qquad j \vdash \mathsf{done} \qquad j \vdash \pi \qquad j \vdash x$$

$$\frac{j \vdash e}{j \vdash \mathsf{flat}(e)} \qquad j \vdash \mathsf{behavior/c}\ (\kappa m \vee \dots \vee \kappa m)$$

$$\frac{j \vdash e}{j \vdash \mathsf{become}\ e} \qquad \frac{j \vdash e}{k \vdash \mathsf{check}^j\ v\ [\![e]\!]^j} \qquad \frac{j \vdash e}{k \vdash \mathsf{mon}_l^{j,k}\ \kappa\ [\![e]\!]^j}$$

Fig. 12. Well-formedness judgment

7.2 Ownership Propagation

Ownership information is propagated automatically during the execution of the program. We augment our semantics to keep track of this ownership information, which is done solely for the purpose of this proof. To this end, a similar approach as Dimoulas et al. is taken.

Only two expressions change the ownership of a value: function applications and send expressions. During a function application, an argument is passed from the party applying the function to the party that owns the function itself. After the argument has been passed, the owner of the function declares itself the owner of the argument and execution proceeds as usual. This semantics is depicted in the rule below. We highlight changes to our semantics in *gray*.

$$[\textsc{App}] \quad E[\ [\![\ \lambda x.e\ ]\!]^{l'}\ v] \longrightarrow E[\ [\![\ [x \mapsto [\![v]\!]^{l'}]e\ ]\!]^{l'}\ ]$$

A send expression causes a similar change in ownership. Whenever a value is sent as part of the payload of a message, its ownership is transferred from the sender of the message, to the owner of the actor reference. It is important to note that a potential receiver contract and contract on the payload have already been checked. Therefore, ownership only moves whenever a fully checked message is being delivered. The updated rules for *enhanced message sends* are depicted below:

$$\begin{aligned}[\textsc{E-Send}]\ & \{\langle \pi_1, E[\overline{\mathsf{send}}\ \kappa c\ [\![\pi_2]\!]^{l'}\ \tau\ v], t_1, M_1\rangle, \langle \pi_2, e, t_2, M_2\rangle\} \uplus \mathcal{A} \\ & \longrightarrow_{\mathcal{A}} \{\langle \pi_1, E[\mathsf{nil}], t, M\rangle, \langle \pi_2, e, t, (\tau, \kappa c, [\![v]\!]^{l'}) \cdot M\rangle \\ [\textsc{E-SendSelf}]\ & \langle \pi, E^l[\overline{\mathsf{send}}\ \kappa c\ [\![\pi]\!]^{l'}\ ], t, M\rangle \longrightarrow_{\alpha} \langle \pi, E^l[\mathsf{nil}], t, (\tau, \kappa c, [\![v]\!]^{l'}) \cdot M\rangle\end{aligned}$$

Having defined how ownership propagates when a message is sent, we discuss how contract monitors on the receiver of the send expression propagate ownership labels. This contract monitoring is defined by the stacking rules, which we adapt accordingly.

$\boxed{\text{Value-based contracts on the payload}}$

$$\begin{aligned}&\mathsf{stack}_p\ [\![\ (\mathsf{mon}_l^{j,k}\ \kappa_1\ v_1)\ ]\!]^k\ v_2 = (\mathsf{stack}_p\ v_1\ (\mathsf{mon}_l^{k,j} \kappa_p\ [\![\ v_2\ ]\!]^k)) \\ &\textbf{where}\ \ \kappa_1 = \mathsf{behavior/c}\ \kappa m_1 \vee \dots \vee \kappa m_n \qquad \mathsf{message/c}\ \tau\ \_\ \kappa_p\ \_ \in \{\kappa m_1, \dots, \kappa m_n\}\end{aligned}$$

$\boxed{\text{Receiver contracts}}$

$$\begin{aligned}&\mathsf{stack}'_r\ [\![\ (\mathsf{mon}_l^{j,k}\ \kappa_1\ v_1)\ ]\!]^k\ c = \mathsf{stack}'_r\ v_1\ (\lambda x.\lambda y.\ [\![\ (check^k\ (\kappa r\ y)\ (c\ [\![\ x\ ]\!]^k\ y)))\ ]\!]^k \\ &\textbf{where}\ \ \kappa_1 = \mathsf{behavior/c}\ \kappa m_1 \vee \dots \vee \kappa m_n \qquad \mathsf{message/c}\ \tau\ \kappa r\ \_\ \_ \in \{\kappa m_1, \dots, \kappa m_n\}\end{aligned}$$

For contracts on the payload the change is straightforward. Monitoring the payload value with a valued-based contract causes the value to move under the ownership of the monitored actor reference. The transformation is similar for receiver contracts. It is important to note that we chose the client party k as the ownership annotation for the stacking rules. This is not a coincidence, since the negative party of the contract monitor must align with ownership label. In Section 7.3 we show that only this combination is possible. All other rules, except for enhanced receives, do not cause any changes in ownership, and simply propagate the labels as is.

The last part of the contract language we need to discuss are the communication contracts, which are monitored by monc expressions. Similar to the well-formedness statements about the mon expression, we formulate a well-formedness statement for the monc expression. To this end, we extend our judgment $\vdash$ with an additional rule: $k \vdash e \Rightarrow j \vdash \mathsf{monc}^k v_\kappa \; [\![e]\!]^k$

Recall that the monc expression lacks any client or contract label. The reason is that the communication contract always monitors the communication effects of the *server* party (or any of its transitive communication effects). This yields an interesting well-formedness judgment where only the inner expression in the contract monitor has an ownership annotation, and is related to the label on the contract monitor. In terms of propagation, we adapt the stacking rules for communication contracts to propagate the ownership of the owner of the monitored actor reference (depicted below). The ownership of the resulting contract monitor does not change during the execution of the program. This point is pivotal for our correctness proof since the well-formedness judgment needs to hold whenever a contract monitor expression is reduced.

$$\mathsf{stack}_c(\mathsf{mon}_l^{j,k} \; \kappa_1 \; [\![\; v_1 \;]\!]^j \;) = [\![\; \lambda x.\mathsf{monc}^j \; \kappa c \; x \;]\!]^j \circ \mathsf{stack}_c$$

Note that, again, the stacking rules assume that the ownership label j of the monitored entity corresponds to the server label j of the contract monitor. We show in Section 7.3 that this assumption always holds when the program semantics reaches the reduction of the stacking rules.

We introduced four-tuples to represent the messages that are intercepted by the communication contract monitor. These four-tuples can also be annotated with ownership annotations. We argue that if a message is annotated with a certain ownership label l, its constituents are also owned by that same party. Therefore we define $[\![(e_\kappa, v_1, \tau, v_2)]\!]^l$ to mean $([\![e_\kappa]\!]^l, [\![v_1]\!]^l, \tau, [\![v_2]\!]^l)$.

7.3 Blame Correctness Theorem

Our blame correctness theorem is split into two cases. The first case deals with the sequential subset of the language and with contracts on the interface of the actor. The second case deals with communication contracts.

Definition 7.3. Given an expression e, an actor system $\mathcal{A}$, and an owner j_0 for which the judgment $j_0 \vdash e$ holds, the contract system $\rightarrow_\kappa$ is **blame correct** iff for any reduction from $\mathcal{A}$ that leads to a contract check $\mathcal{A} \rightarrow_\kappa^* \{\langle \pi, E[\mathsf{check}^j \; \kappa \; e], t, M \rangle\} \uplus \mathcal{A}'$ the following holds: $e = [\![v]\!]^j$.

Put differently, an actor system is *blame correct* if and only if for any reduction leading to a contract check, the blame label aligns with the ownership of the value being checked. We now present our main theoretical result: blame correctness for the contract system.

Theorem 7.4. *$\rightarrow_\mathcal{A}$ is blame correct.*

Proof. The key insight is that the mon state described in Definition 7.3 is a well-formed actor system. The idea of the proof is to show that any state of the actor system will eventually resolve to a well-formed actor system before reducing to a check expression. First, realize that a check expression, according to our semantics, can only originate from mon expressions of the form $\mathsf{mon}_l^{j,k} \; \kappa \; e$. Therefore, we have to show that for any initial program and intermediate state, every contract monitor is either well-formed, or always reduces to one that is well-formed. We do so by case analysis on the reduction rules. Essentially, the case analysis comes down to four rules [MonFun] and [E-Send], [E-SendSelf] and [M-Send].

- [MonFun] reduces a higher order contract monitor to a function that includes the necessary monitoring code. Thus, this new function contains monitor expressions that do not satisfy the well-formedness condition. From the definition of our semantics we have:
$$\mathsf{mon}_l^{j,k}\ (\kappa_1 \to \kappa_2)\ [\![v]\!]^j \to [\![\lambda x.\mathsf{mon}_l^{j,k}\ \kappa_2\ ([\![v]\!]^j\ \mathsf{mon}_l^{k,j}\ \kappa_1\ x)]\!]^k$$
Neither the monitor for κ_2, nor for κ_1, are well-formed as they both lack ownership labels. However, for these monitors to be reduced the λ-expression has to be applied. This application causes variable x to be substituted for a value owned by k, rendering the monitoring expression for κ_1 well-formed. The contract monitor for κ_1 then reduces to a value, or a blame error. In the case of a value, the value is passed to the function v, which is owned by j, which causes the value to be owned by j as well. The second monitor is now well-formed, which concludes the proof.
- [E-Send], [E-SendSelf] the sending rules reduce an enhanced send expression to nil and cause the message to be added alongside its contract to the receiving actor's mailbox. If a message is sent to a monitored actor reference, the stacking rules (cf. Fig. 7) generate contract monitors for its payload, receiver, and communication contracts. Assuming that the contract monitor is owned by the client label of the monitor (which we show in Lemma 7.5), the generated contract monitors on the payload and receiver are well-formed by definition and do not require any further analysis.
- Rule [M-Send] introduces a mon expression through its return value and the dosend meta-function. We must show that its argument v_2 can only be owned by j. The value v_2 originates from the argument of the monc expression which is a representation of the monitored message as a tuple. Recall that we defined earlier that the ownership of the tuple is recursively propagated to the ownership of its constituents. Therefore, to show that our blame correctness theorem holds for [M-Send] rules we have to show that the tuple $(e_\kappa, v_1, \tau, v_2)$ is owned by party j (proved in Theorem 7.6) □

Lemma 7.5. *$[\![mon_l^{j,k}\ \kappa\ v_1]\!]^k$ always holds when reduction reaches a* $\mathsf{stack_p}$ *rule.*

Proof. The proof for [MonFun] and [M-Send] are similar to our previous theorem. We prove the case for [E-Send] and [E-SendSelf] by induction on program reduction steps.

- *Base case.* Contract monitors in stack_p are monitored actor references. This can only occur in the beginning of a program if the source code contains such a contract monitor. By pre-condition of our previous theorem, this monitor is well-formed.
- *Induction case.* Assuming that the lemma holds for all previous steps of the reduction, we show that it holds for the next reduction of [E-Send] and [E-SendSelf]. Indeed, we can show that [MonFun] and [M-Send] produce well-formed contract monitors, and that the base case of [E-Send] and [E-SendSelf] also produces valid contract monitors. Then, by definition, [E-Send] and [E-SendSelf] produce well-formed contracts (by similar argument as in the previous theorem) and therefore any value flowing into the first argument of stack_p must also be well-formed. □

We proceed with the second part of our theorem. To show the blame correctness of communication contracts, we have to show that a party is only blamed when the message is owned by that party.

Theorem 7.6. *Our communication contracts are blame correct. Given a program e_0 and owner l_0 and a reduction $\{\langle \pi, e_0, \emptyset, \emptyset\rangle\} \twoheadrightarrow_{\mathcal{A}} \{\langle \pi, E[\mathsf{monc}^j\ \kappa c\ v], t, M\rangle\} \uplus \mathcal{A}'$ then $v = [\![v_0]\!]^j$*

Proof. mon$c$ expressions are only introduced in stack$_c$ rules and during the reduction of the mon$c$ expression itself. We consider both cases separately.

- stack$_c$: When introduced in the stacking rule, a λ-expression of the form $[\![\lambda x.\mathsf{mon}c^j\ \kappa c\ x]\!]^j$ is produced. Here, the ownership propagation rules for function applications apply, moving the ownership of the argument to the owner of the function. This means that the monitored message or trace (which are treated as values) will always be owned by j, thereby satisfying the theorem.
- mon$c$: Since mon$c$ expressions can only be introduced in stack$_c$ expressions, any λ-expression originating from the reduction rules for mon$c$ will be owned by the owner of the λ-expression in the stack$_c$ rule. The lambda originating from the mon$_c$ rule will hence also be owned by j, thereby satisfying our theorem. □

8 Related Work

Actors and processes. Whereas our work targets the actor model, much of the related work centers around the π-calculus, where processes are anonymous and communication is achieved through bi-directional message channels. Even though both models have their own distinct features, there is a correspondence between them. Fowler et al. [2017] discuss this correspondence and uncover an isomorphism between the two models. Their isomorphism is achieved through a translation between the two calculi. They find that the actor model is straightforward to simulate in the $\pi-$calculus, whereas the translation from the $\pi-$calculus to the actor model is more involved. Nonetheless, this shows that both models are equivalent and can be used to simulate the other. Therefore, our contract system can also be applied to the π-calculus but would have to be adapted to take the bi-directionality of its channels into account. We opted to use the actor model as a basis since it requires less infrastructure as it models communication using messages to actor references, rather than establishing bi-directional channels.

Type systems. (Multi-)party session types [Honda et al. 2016] are closely related to our communication contracts. However, they differ in a few ways. First, traditional session types are limited to predicates that can be decided *statically*. Second, session types describe sessions between two or more parties (cf. multiparty session types). In each of these sessions, participants have a fixed role and therefore also a fixed type. The process that is participating in the session cannot handle other sessions simultaneously. Actors, in contrast, do not belong to any particular session, and messages from different sessions might be interleaved. This enables developers to define complex architectures that include load balancers, circuit breakers, ... These components do not belong to a particular session and would break if this were attempted.

Another difference is that session types are usually formulated in the context of the process calculus. In this calculus, processes establish explicit *bi-directional channels* that are used for communicating between process. A process may have multiple channels in scope simultaneously. This difference is not substantial since one communication model can easily be translated to the other [Fowler et al. 2017]. Even more so, session types have also been applied to actor systems [Neykova and Yoshida 2017]. However, in this setting, actors are still assigned a role which is used to initiate and maintain communication throughout a session. In our actor and contract system, actors do not necessarily need to keep the same role throughout their lifetime and can assume different roles simultaneously.

Contract Systems. Harnie et al. [2010]; Scholliers et al. [2015] propose a contract system for AmbientTalk [Cutsem et al. 2014] called *computational contracts*. AmbientTalk is a programming language that is based on the principle of *communicating event loops*. In this model, actors consist of multiple objects that can be individually addressed from other actors through far references. Thus, a message does not only contain the intended actor, but also the intended object. Scholliers et al. propose a contract system on top of this model. Their contract system expresses constraints over objects and over the computational behavior of their methods. Therefore, their contract language is similar to ours in the sense that we also express constraints over the actors in the system and their message handlers. However, the contract language of Scholliers et al. lacks a way to express the intended receiver of a message (which we address through receiver contracts) and has no concept of contract chaining, thus making it more difficult to express contracts over communication chains that span multiple actors.

Disney et al. [2011] propose a contract system which they call *temporal contracts*. Their contract system expresses constraints on the *temporal behavior* of functions in a sequential programming language. This temporal behavior includes function calls and returns. Their system is *higher-order* in the sense that temporal contracts can be attached to arguments of functions or return values. Our contract system is higher-order in the same sense, message contracts can introduce communication contracts on actor references that were not monitored before. However, in contrast to the work by Disney et al., our contract system supports concurrent processes and can also put communication contracts on the receiver of a message, which is equivalent to putting a contract on the callee.

Waye et al. [2017] propose a contract system for modern web services called Whip. The proposed contract system is similar to our contracts on the interface of an actor, but does not offer communication contracts nor contracts on the receiver. The system is also higher-order in the sense that services can take and return references to other services which can also be monitored by contracts. Their focus is primarily on the practical applicability of such a system. In contrast to our contract system, theirs treats services as black boxes, enabling different programming languages to be used for their implementation. To this end, they introduce the concept of a *service adaptor* that functions as a contract monitor independent from the monitored services, such that they are unaware of contract monitoring. We do not aim for such transparency in this paper. Nonetheless, our communication contracts are only interested in the *communication effects* of the message handler they are monitoring. Therefore, the contract monitor is not concerned with the internal state of the actor and could be implemented as a separate entity.

Gommerstadt et al. [2022] propose *session-typed concurrent contracts*. Their contracts are constructed as *partial identity processes*. Meaning that their contracts either allow the message to be passed through as is, or block the message entirely. This is problematic in a higher-order setting, where actor references can be sent as part of message payloads. In such settings, the values in the payload of the outgoing message can be changed to include contract monitors, therefore violating the partial identity property. Thus in order for their system to support higher-order languages, their monitors have to be applied manually. Another interesting aspect of their approach is that their contracts are represented as processes. This allows their contracts to be stateful since each process in the system can have an internal (private) state. Our system does not support stateful contracts, and stays closer to the traditional treatment of contracts, but could be extended to support them without violating our blame correctness property. Finally, the contract system by Gommerstadt et al. limits its contracts to a single channel, and piggybacks on the session-types for expressing constraints on communication that spans multiple actors. We argue that this makes our contract system more powerful, since it allows for expressing much more dynamic contracts. For example, our contracts can change the expected communication behavior across a communication chain based on the contents of the payload, or based on previous communication behavior.

Contracts have also been used to define higher-order sessions. Melgratti and Padovani [2017] propose a contract system that follows the shape of processes in the process calculus. Their contracts are also *dependent* and can express properties in terms of earlier messages. Since channels in the process calculus are bi-directional, their contracts express constraints on both the values received and sent by the receiving process. Their contracts are therefore a subset of our contracts where the receiver of each message is predetermined by the direction of the channel. Our contract system in contrast, allows for arbitrary receivers that are specified using receiver contracts.

In summary, current contract languages for distributed systems are primarily focussed on the interface of the actor and only allow expressing properties about the tag and payload the message. Those that support behavioral contracts do so on a single actor, while our contracts, due to their recursive structure, allow expressing behavioral properties about the receiver of messages in the entire communication chain.

9 Limitations & Future Work

Blame Propagation and Recovery. In concrete implementations, blame labels usually correspond to modules in the source application. Therefore, parties correspond to regions in the code that are to blame for the contract violation. However, the blame error might be produced in different part of the application that does not correspond to any region indicated by the blame label. This is not only true for our contract system but also for any higher-order contract system. Higher-order functions become wrapped with contract monitors which can be passed to unmonitored parts of the code. Whenever these higher-order contract monitors are applied, a blame error occurs in a part of the application that is not covered by the blame label. The impact of this problem is minimal for sequential languages since one failure causes the entire application to halt. In a concurrent or distributed setting however, processed or actors execute independently from each-other and are preferably kept online. In this paper, we focussed on the design and formalisation of a contract language to support encoding communication patterns within an actor system. We consider *blame propagation and recovery* as an orthogonal problem that can be answered in future work.

Types of communication contracts. In this paper, we presented two types of communication contracts: *ensures* and *only* contracts. We realize that the design space for these communication contracts is considerably larger than the two variants we presented here. However, we argue that our blame assignment semantics would remain identical and equally valid for the remaining variants in this design space.

10 Conclusion

In this paper, we introduced a novel contract system to express constraints on the communication effects of actors in an actor system. This contract system includes a rich support for defining the interface of an actor in the system as a set of message contracts. Next to contracts about the message tag and its payload, our contract system also supports expressing properties about the recipients of messages and contracts on their communication effects. We introduced two types of communication contract. The first type ensures that all messages from a set of messages are sent during an actor's turn. The second type limits the messages that can be sent to those that are specified in the communication contract. A defining feature of our message contracts is that they are recursively structured. Indeed, a message contract contains a communication contract which can again include a message contract. This recursive structure enables specifying *communication chains* spanning many different actors in the actor system. It puts blame at the component at the start of this chain essentially saying that the start of the chain is at fault for initiating a faulty chain of messages. This gives the programmer more freedom compared to traditional contract and session types systems which typically let blame boundaries align with actor boundaries.

We formalized this contract system on top of the classic actor model. Using this formalisation we have proven that our blame assignment is correct with respect to ownership. Essentially, the proof shows that a party is only to blame whenever the value or message causing the contract violation actually originated from that party.

References

Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT press.

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6269)*, Paul Gastin and François Laroussinie (Eds.). Springer, 162–176. https://doi.org/10.1007/978-3-642-15375-4_12

Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.* 40, 3-4 (2014), 112–136. https://doi.org/10.1016/J.CL.2014.05.002

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 215–226. https://doi.org/10.1145/1926385.1926410

Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 117–131. https://doi.org/10.1145/2951913.2951930

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 214–233. https://doi.org/10.1007/978-3-642-28869-2_11

Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal higher-order contracts. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 176–188. https://doi.org/10.1145/2034773.2034800

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59. https://doi.org/10.1145/581478.581484

Simon Fowler, Sam Lindley, and Philip Wadler. 2017. Mixing Metaphors: Actors as Channels and Channels as Actors. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017 (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:28. https://doi.org/10.4230/LIPICS.ECOOP.2017.11

Teodoro Freund, Yann Hamdaoui, and Arnaud Spiwack. 2021. Union and intersection contracts are hard, actually. In *DLS 2021: Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages, October 19, 2021*, Arjun Guha (Ed.). ACM, 1–11. https://doi.org/10.1145/3486602.3486767

Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2022. Session-typed concurrent contracts. *J. Log. Algebraic Methods Program.* 124 (2022), 100731. https://doi.org/10.1016/J.JLAMP.2021.100731

Dries Harnie, Christophe Scholliers, and Wolfgang De Meuter. 2010. Ambient Contracts. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 28 (2010). https://doi.org/10.14279/TUJ.ECEASST.28.397

Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. 2021. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). 10:1–10:30. https://doi.org/10.4230/LIPICS.ECOOP.2021.10

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 273–284. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/1328438.1328472

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and blame assignment for higher-order session types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 582–594. https://doi.org/10.1145/2837614.2837662

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 285–296. https://doi.org/10.1145/2103656.2103691

Roland Kuhn, Brian Hanafee, and Jamie Allen. 2017. *Reactive Design Patterns* (1st ed.). Manning Publications Co., USA.

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 517–530. https://doi.org/10.1145/2872362.2872374

Hernán C. Melgratti and Luca Padovani. 2017. Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.* 1, ICFP (2017), 35:1–35:29. https://doi.org/10.1145/2384616.2384685

Bertrand Meyer. 1998. Design by Contract: The Eiffel Method. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 446. https://doi.org/10.1109/TOOLS.1998.711043

Robin Milner. 1999. *Communicating and mobile systems - the Pi-calculus*. Cambridge university press.

Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Log. Methods Comput. Sci.* 13, 1 (2017). https://doi.org/10.23638/LMCS-13(1:17)2017

Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2015. Computational contracts. *Sci. Comput. Program.* 98 (2015), 360–375. https://doi.org/10.1016/J.SCICO.2013.09.005

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 943–962. https://doi.org/10.1145/2384616.2384685

Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. 2024. *Blame-Correct Support for Receiver Properties in Recursively-Structured Actor Contracts (Artifact)*. https://doi.org/10.5281/zenodo.12659179

Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: higher-order contracts for modern services. *Proc. ACM Program. Lang.* 1, ICFP (2017), 36:1–36:28. https://doi.org/10.1145/3110280

Received 2024-02-28; accepted 2024-06-18