ICE 2022 Pre-Proceedings

Clément Aubert Cinzia Di Giusto Larisa Safina Alceste Scalas

13th June 2022

This document contains the *informal* pre-proceedings of the 15th Interaction and Concurrency Experience (ICE 2022). The post-proceedings will be published on EPTCS.

Contents

The Right Kind of Non-Determinism: Using Concurrency to Verify C Programs with Underspecified Semantics

Eduard Kamburjan
University of Oslo, Oslo, Norway
eduard@ifi.uio.no

Nathan Wasser
Sharpmind, Frankfurt, Germany
nate@sharpmind.de

We present a novel and well automatable approach to formal verification of C programs with underspecified semantics, i.e., a language semantics that leaves open the order of certain evaluations. First, we reduce this problem to non-determinism of concurrent systems, automatically extracting a distributed Active Object model from underspecified, sequential C code. This translation process provides a fully formal semantics for the considered C subset. In the extracted model every non-deterministic choice corresponds to one possible evaluation order. This step also automatically translates specifications in the ANSI/ISO C Specification Language (ACSL) into method contracts and object invariants for Active Objects. We then perform verification on the specified Active Objects model, using the `Crowbar` theorem prover, which verifies the extracted model with respect to the translated specification and ensures the original property of the C code for all possible evaluation orders. By using model extraction, we can use standard tools, without designing a new complex program logic to deal with underspecification. The case study used is highly underspecified and cannot be handled correctly by existing tools for C.

1 Introduction

Verification of programs relies on the availability of a formal, or at least a formalizable, semantics of the used programming language. However, the semantics of mainstream programming languages contain challenges that require special attention from programmers and verification tools alike.

In this work we consider the semantics of the C language, which in addition to fully specified behavior contains *undefined*, *unspecified* and *implementation defined* behavior: these semantics describe not exactly what should happen, but leave crucial decisions to the implementing compiler and/or the runtime environment. Our focus here is on the unspecified evaluation order within the C standard, which we refer to as *underspecified*. Importantly, the semantics for underspecified behavior is not *undefined*, as the semantics limits the possible choices. This is not merely a fringe case, but is observable already in natural and small programs. Consider the C program in Fig. 1. The C99 standard [23] does not specify the order of evaluation of the subexpressions in the addition.[1] Indeed, the two main compilers for C return different values: `gcc` 7.4.0 returns 2 (evaluating the second summand first), `clang` 6.0.0 returns 1 (evaluating the first summand first). The reason is that `gcc` uses a stack-based translation of expressions, while `clang` uses a queue-based one.

Verification of underspecified C code is still an open problem and merely *fixing* the choice is not enough for verification: As the semantics is underspecified, compilers are not required to be consistent in their choice *even during the run of a single program* and optimizations are not obligated to preserve the choice of the compiler.

[1]This unspecified evaluation order is also prevalent in other C standards.

```c
int x;
int id_set_x(int val){
  x=1;
  return val;}
int main(void){
  x=0;
  return x + id_set_x(1);}
```

Figure 1: Addition with side-effect.

This effect is further amplified from a software engineering perspective, when program equivalence becomes a problem: For one, changing, updating the compiler, or indeed barely changing its parameters may result in different program behavior. For another, reengineering legacy software, a critical activity to, e.g., enable parallelization [18] cannot rely on analyses proving functional equivalence, if these analyses are not considering underspecification. Before attempting to prove program equivalence, one must be able to reason about functional behavior of programs in a language with underspecified semantics.

Approach. At the core of this work is the idea to transform non-determinism in sequential programs arrising due to *underspecification* to non-determinism due to *concurrency* and then use tools to specify and verify concurrent behavior, which are more advanced and investigated in more detail. Each possible evaluation order is one possible interleaving order.

More precisely, this work presents an approach to *automatically* verify functional behavior of C programs with underspecified semantics, which is based on reducing *underspecification* to *non-determinism* in a fully specified language: We are able to verify functional properties of C programs without undefined behavior with respect to every possible standard-compliant semantics. In this work we build upon the model-extraction approach by Wasser et al. [37] for a subset of the C language and give an *implemented* system that verifies the functional behavior of the extracted model. The extracted model gives a *fully formal* and analyzable semantics for C in terms of an Active Object framework.

We translate C code into an *Active Objects* language [13] and regard sequential C programs as parallel programs, in which the non-determinism arises from parallelism and not from underspecified semantics. Conceptually, this is a rare case where a problem of *sequential* programs is transformed to a problem of *parallel* programs, because the support for analysis of parallel systems is better than the support for reasoning about underspecified semantics.

For Active Objects there are program logics [26] that enable modular reasoning and we are able to employ method contracts for asynchronous calls [27]. The expected behavior under all possible semantics is annotated with ACSL [8] and automatically translated into cooperative contracts and object invariants of Active Objects. Using this approach we give a case study to verify that a highly underspecified recursive function that computes the nth Fibonacci number in one semantics returns a value between 1 and the nth Fibonacci number in every standard-adhering semantics.

Contributions. Our contributions are (1) an implemented approach to *automatically* verify functional behavior of C programs with underspecified semantics, and a deductive verification case study of underspecified C code which is (2) the biggest verification case study of such code that cannot be handled by existing approaches (see next section) (3) the biggest deductive verification case study for Active Objects (in lines of code) to date. The case study can be proven *fully automatically*. Additionally to the conceptual approach and case study, we also contribute a translation of ACSL specifications for C into BPL specifiations for ABS.

State-of-the-Art. Underspecified (and to a lesser degree undefined) semantics are a rarely approached challenge for deductive verification. Here, we review the tools that consider these kinds of semantics.

Frama-C [12] can find (some) *undefined* behavior related to read-write or write-write accesses *between* sequence points. However, it does not recognize *unspecified* behavior when these accesses occur *indeterminately sequenced* as in our examples here, instead only examining a single fixed evaluation order[2] [10, p.40]. Further, while most of ACSL is utilized in Frama-C, this does not include global invariants, which we are able to handle. Additionally, new tools must be built specifically for the C intermediate representation only used within Frama-C, while our approach can profit from all tools available for ABS, which has included so far model checking, simulation, deadlock analysis and deductive verification. RV-Match [1]—based on C semantics formalized [2, 19] in the K framework [3, 36]—is able to find (some) *undefined* and *implementation defined* behavior in C programs, but like Frama-C chooses only a single evaluation order when faced with *underspecified* behavior. This in turn prevents both from finding undesired behavior that is only obvious when a different evaluation order is chosen. While our approach currently works only with an admittedly smaller subset of C containing underspecification than that allowed in RV-Match and Frama-C, it faithfully considers all possible evaluation paths allowed by the standard. Cerberus [4, 33] is an analysis tool for undefined and underspecified behavior; however, it cannot utilize any specifications and its treatment of unspecified evaluation order of *side effects* does not match the C standard, as demonstrated in [37]. The separation logic system of Frumin et al. [17], based on small-step semantics in Coq [30] correctly treats underspecification. They give a formal system to verify a program in their toy language λMC and check effects of underspecified behavior with a modified separation logic. In contrast to the subset of C we consider, λMC is emphatically *not* a subset of C and is described as merely a C-style language[3]. Verification of *any* C program therefore requires manual translation into an equivalent λMC program and manual specification of the λMC program in Coq. Our model-extraction based approach is fully automated, can be used with standard program logics and analyses for Active Objects and does not rely on complex rule modifications to handle underspecified behavior. We stress that this automation includes the verification, which needs not be performed by the user in an interactive prover such as Coq [5].

Holzmann and Smith [22] attempt to reuse the SPIN model checker by extracting Promela code from a C program. However, their approach requires *manual* translation/adjustment (flattening) of the underspecified parts. Furthermore, Promela/SPIN only support model checking and cannot be applied to unbounded inputs. Concerning semantics, several formalizations [16, 34, 35] of the C semantics deal with underspecified evaluation order without giving a reasoning system.

To conclude the overview of the state-of-the-art, there is no satisfying approach to verify underspecified C code and the partial approaches are not suited for automatization.

Structure. In Sec. 2 we investigate the program in Fig. 1 in more detail. In Sec. 3 we give preliminaries: the basics of ABS [24], the Active Object language used, and its contracts. In Sec. 4 we describe the model-extraction, which we then use in Sec. 5 to verify the Fibonacci case study. We conclude in Section 6. The accompanying technical report with formal details, proofs and a link to the implementation is not referred to for the double-blind review.

[2]E.g., value analysis in Frama-C claims that the program in Fig. 1 can only return 2.

[3]Even this is debatable, but underspecified C-style behavior is present.

2 Overview over Workflow

Before we introduce the used systems, we illustrate our approach using the code in Fig. 2, which adds ACSL specifications to the previous example. The strong global invariant specifies a condition that must hold at every point during execution, while the requires/ensures clauses are standard pre/postconditions.

Specified C-code is translated into specified ABS-code. ABS is object-oriented and uses the following concurrency model: (1) An object cannot access the fields of another object. (2) Every method call is asynchronous (i.e., does not block the caller) and returns a future. A future can be used to synchronize on the called method and read its eventual return value. (3) Only one process is active per object and a process can only be interrupted when executing an **await** `g` statement. An **await** `g` statement waits until all futures in the guard `g` are resolved, i.e., their process has terminated. There are no global variables and for specification, ABS supports object invariants and method contracts.

The code in Fig. 3 shows a (prettified) part of the translation of Fig. 2. The global variables are handled by a special (singleton) class `Global`. In `Global`, each global variable is a field and the global invariant becomes the object invariant of this class. Similarly, the global invariant is also added as pre/postcondition to the setter and getter method handling the fields.

Each C-function `f` is translated into an ABS-class `C_f` and an interface `I_f` with a `call` method that models its execution. The function contract of `id_set_x` becomes the method contract of `I_id_set_x.call`. We only show the translation of **main** in detail. Again, the function contract becomes the method contract of `call`. The other methods in the class `C_main` model memory accesses to global variable `x`, calling function `id_set_x` and addition with the `+` operator.

The `call` method is a translation of the **main** function. It first sets `x` to 0 and than waits for this operation to finish — the **await** at line 21 models synchronization at the sequence point `;`. The next three lines translate the addition operation and contain *no* **await**, because the C-expression contains no sequence point. The two calls to model evaluation of the subexpressions are called in one order, but may be executed in a different one.

The method `op_plus_fut_fut` models evaluation of the addition expression. It takes two futures, i.e., two references to *yet unfinished executions*. It then synchronizes with both of them, i.e., it waits until *both* are resolved (line 34) and then adds the corresponding return values. It depends on the global scheduling which method is executed first and therefore whether the read triggered in `C_main` or the write in `C_id_set_x` takes place on `Global` first. Note that the specification of `C_main` is also automatically derived from the ACSL specification. The translated model can now be passed to the `Crowbar` verification system, which checks that the code adheres to its specification. It indeed does so and, as expected, fails to close the proof if the specification is wrong, i.e., if the results is specifed as only 1 or only 2.

```
int x; //@ strong global invariant x == 0 || x == 1;
int id_set_x(int val)
/*@ requires val == 1; ensures \result == 1; @*/ {
  x=1; return val;}
int main(void)
/*@ ensures \result == 1 || \result == 2; @*/ {
  x=0; return x + id_set_x(1);}
```

Figure 2: Specified addition with side-effect.

```
[Spec:ObjInv(this.x == 0||this.x == 1)]
class Global implements Global {
 Int x = 0;
 [Spec:Ensures(result == 0||result == 1)]
 Int get_x() { return this.x; }
 [Spec:Requires(value == 0||value == 1)]
 Unit set_x(Int value) { this.x = value; }
}
class C_id_set_x(Global global)
      implements I_id_set_x {
 [Spec: Requires( val == 1 )]
 [Spec: Ensures( result == 1 )]
 Int call(Int val){...}// executes id_set_x(val)
 ... }
class C_main(Global global)
      implements I_main {
 [Spec:Ensures(result == 1||result == 2)]
 Int call() { // executes main()
  Fut<Unit> tmp_4 =
    this!set_global_x_val(0); // sets x to 0
  await tmp_4?; // introduces sequence point ‘‘;’’
  Fut<Int> tmp_5 =
    this!get_global_x(); // reads x
  Fut<Int> tmp_6 =
    this!call_id_set_x_val_0(1);//calls id_set_x
  Fut<Int> tmp_7 =
    this!op_plus_fut_fut(tmp_5, tmp_6);//add
  await tmp_7?; // introduces sequence point ‘‘;’’
  return tmp_7.get; // returns result of addition
 }
 [Spec: Ensures(valueOf(fut_arg1) + valueOf(fut_arg2) == result)]
 Int op_plus_fut_fut(Fut<Int> fut_arg1,
                     Fut<Int> fut_arg2) {
  await fut_arg1? & fut_arg2?;
  Int arg1 = fut_arg1.get;
  Int arg2 = fut_arg2.get;
  return ( arg1 + arg2 );
 }
 ... }
```

Figure 3: Partial translation of Fig. 2.

3 Active Objects and Their Verification

In this section we give the preliminaries for our work: the ABS language and cooperative contracts. For space reasons, we refrain from introducing the full formalisms and refer to [26] for a full definition of the underlying program logic and to [27] for a definition of the used ABS semantics and cooperative contracts. We stress, however, that the approach is fully formal.

ABS [24] is an executable, object-oriented modeling languages based on Active Objects [13], designed to model and analyze distributed systems. It has been applied to model a wide range of concurrent software systems, such as cloud-based services [14, 31], YARN [32] or memory systems [28].

Overview. ABS syntax is largely based on Java and we refrain from describing the full language here. Instead, we introduce ABS in an example-driven way to demonstrate its concurrency model and formal semantics. The main features of the concurrency model can be summarized with the points below:

Strong Encapsulation. Every object is strongly encapsulated at runtime, such that no other object can access its fields, not even objects of the same class.

Asynchronous Calls with Futures. The ABS language combines actors [21] with futures [6]. Each method call is asynchronous and generates a future. Futures can be passed around and are used to synchronize on the process generated by the call. Once the called process terminates, its future is *resolved* and the return value can be retrieved. We say that the process *computes* its future.

Cooperative Scheduling. At every point in time, at most one process is active in an object. Active Objects are preemption-free: A running process cannot be interrupted unless it *explicitly* releases control over the object. This is done either by termination with a **return** statement or with an **await** `g` statement that waits until guard `g` holds. A guard polls a set of futures and holds iff all futures in it are resolved.

These features ensure that a process has exclusive control over the heap memory of its object between syntactically marked statements. This vastly simplifies deductive verification, as between such statements techniques from sequential program verification carry over directly.

Example 1. *As the extracted models from C code are rather unintuitive, we demonstrate the concurrency model of ABS with a more natural program.*

Fig. 4 gives an ABS model with two objects that folds some binary operation over three numbers: one object that performs the operation and a second object that performs the folding. Interface `Fold` *defines an interface for the fold. Lines 2 and 3 give the specification, which we discuss in more detail below. Here, we specify that the input values must be positive (*`Requires`*) and that the result is positive (*`Ensures`*). Interface* `Comp` *specifies a single method, which performs some operation that also operates only on positive numbers. Class* `FoldC` *implements the folding and has a field* `comp` *that points to a* `Comp` *instance. We specify that the field is initialized with a non-null value (*`Requires`*) and stays non-null (*`ObjInv`*). It has a field* `last` *to store the intermediate result. ABS uses a main block to initialize the system, which here creates one instance of each class, starts two* `fold`*-processes and synchronizes on both. There is no await in the class – the processes executing* `C.fold` *do not overlap, so the value of* `last` *cannot change before it is returned and it is safe to save the intermediate value in this field.*

Cooperative Method Contracts. Here, we give the used fragment of the specification language for ABS: cooperative method contracts [27] and object invariants for Active Objects [15]. We recap the Behavioral Program Logic [26] used to verify cooperative method contracts.

Cooperative Method Contracts use two kinds of preconditions for methods: *parameter preconditions*, which describe the expected parameters; and *heap preconditions*, which additionally describe the class fields. Splitting the precondition is necessary, because the parameters are controlled by the *caller process* (and must be guaranteed by the caller), while the fields are controlled by the last active process in the *callee object* (and must be guaranteed by this process). There are also two postconditions: the heap postcondition defines the final state upon termination of the method in terms of its fields and local variables plus a special program variable `result` for the return value; the parameter postcondition defines the return value in terms of the parameters. The parameter postcondition can be used upon reading from the future if the call parameters are known.

We also use object invariants, which must hold at every point a method loses or regains control over the object: at method start, termination and **await** statements. The initial state of classes is specified with *creation conditions*.

```
interface Fold {
  [Spec: Requires(a>0 && b>0 && c>0)]
  [Spec: Ensures(result>0)]
  Int fold(Int a, Int b, Int c);
}
interface Comp {
  [Spec: Requires(a>0 && b>0)]
  [Spec: Ensures(result>0)]
  Int op(Int a, Int b);
}
class CompC implements Comp { ... }
[Spec: Requires(comp != null)]
[Spec: ObjInv(comp != null)]
class FoldC(Comp comp, Int last)
      implements Fold{
  Int fold(Int a, Int b, Int c){
    Fut<Int> f = comp!op(a, b); last = f.get;
    f = comp!op(last, c); last = f.get;
    return last;
  }
}
{ Comp a = new CompC();
  Fold c = new FoldC(a,0);
  Fut<Int> f1 = c!fold(1,2,5);
  Fut<Int> f2 = c!fold(1,2,4);
  await f1? & f2?; }
```

Figure 4: Simple ABS Model, slightly beautified.

Specification. Method signatures in interfaces may be annotated with parameter preconditions of the form `[Spec:Requires(e)]` and postconditions (`[Spec:Ensures(e)]`), where `e` is an expression of Boolean type. Similarly, method implementations in classes may be annotated with heap pre- and postconditions. A heap precondition that could be a parameter precondition is automatically transformed. Classes may be annotated with object invariants `[Spec: ObjInv(e)]` and creation conditions `[Spec: Requires(e)]`. Loops may be annotated with loop invariants `[Spec: WhileInv(e)]`. The specifications in Fig. 4 are explained in Example 1.

Full cooperative contracts have mechanisms to specify and verify **await** statements with suspension contracts and **get** statements with resolving contracts [27]. Similarly, so called *context sets* [27] are used to specify and analyze the heap preconditions. As neither heap preconditions nor suspension or resolving contracts are used by the extracted models, we refrain from introducing them in detail.

Verification `Crowbar` [29] is a verification system for `ABS` that implements symbolic execution (SE) i.e., the step-wise execution of statements to generate a set of first-order logic formulas. Validity of all generated formulas implies safety of the method. The resulting formulas are output in SMT-LIB [7] format and passed to solvers such as Z3.

Additionally to verifying cooperative contracts, `Crowbar` implements a lightweight deadlock checker for ABS that contrary to existing deadlock checkers for ABS [25, 20], requires no main block: The structural deadlock analysis deduces which methods cannot be part of a deadlock *for any program*: A deadlock is a cycle of dependencies caused by future (and condition) synchronizations [25] and is analyzed in terms of cycles in dependency graphs between synchronizations, objects and methods. Any method that contains no synchronization cannot be part of any dependency cycle, it is *structurally deadlock-free*.

Similarly, all methods that only call deadlock-free methods and synchronize only on their futures are not part of any deadlock.

Example 2. *Consider Ex. 1. If the implementation of* `CompC.op` *contains no blocks or call, e.g., the statement* `return a*b`*, then we can show deadlock freedom.*

`CompC.op` *is structurally deadlock-free: it contains no synchronization or suspension.* `C.fold` *depends only on* `CompC.op` *and is thus not part of any deadlock.*

4 Extraction of Annotated Model

In order to extract an ABS model annotated with appropriate specifications from a (specified) C program, we extend the approach from [37] (which extracts a non-deterministic Active Objects model from C code containing underspecified behavior) by automatically generating some specifications which are sound by construction and generating all other specifications by translation of the specifications in the underlying C program. In order to translate ACSL function contracts into method contracts it was also required to slightly change the manner in which function parameters were modeled, from parameters of the class to parameters of the `call` method within the class. Otherwise, simple functional properties would have required reasoning about heap properties.

ACSL

The *ANSI/ISO C Specification Language (ACSL)* [8] is a behavioral specification language for C programs, used by the state-of-the-art *Frama-C* [12] tool suite. ACSL can be used to specify function contracts (pre- and postconditions), data invariants over global variables and some further constructs, such as loop invariants, statement contracts (pre- and postconditions for a single statement or block of statements), assertions or ghost code.

Function contracts consist of a `requires` clause for the precondition and an `ensures` clause for the postcondition. Both clauses can be simple C expressions of arithmetic type[4], with the postcondition allowed to contain `\result` to refer to the return value. Additionally, an `assigns` clause to specify which locations may be accessed can be given. We ignore `assigns` clauses for now as they are not directly relevant for underspecified semantics.

ACSL allows two types of data invariants on global variables: 1. *strong global invariants*, which hold at all times; and 2. *weak global invariants*, which hold before and after each execution of a function call and can thus equivalently be added as a requires and ensures clause to all functions. We therefore focus here only on strong global invariants, in particular as these cannot be easily dealt with in Frama-C. Furthermore, we restrict strong global invariants to properties about single variables and thus exclude relational properties.

4.1 From C Code to ABS (`C2ABS`)

`C2ABS` [37] is an Eclipse plugin which extracts an ABS model from a C program. Here we describe how this extraction takes place. In the next subsection we describe the novel extension of this model extraction: synthesizing specification annotations for the extracted model. Table 1 details how C concepts are translated into ABS. The basic idea is to have one Active Object which models access to global variables and further model each executed function call as its own Active Object. Within these function

[4]Full ACSL allows more operators, which we ignore for now.

C	ABS
Top-level declarations	
Global variables	Class `Global` with methods to get/set variable values
definition of function `f`	class `C_f` with parameter `global`
Execution of function `f`	Execution of `call` method on object of class `C_f`
Parameters and local variables	
const parameter	parameter of `call` method
non-const parameter	parameter of `call` method stored in field
const local variable	local variable
non-const local variable	field
Local const read	Direct variable/parameter access
Other (sub-)expressions	Methods awaiting parameters and:
global read/write	synchronous call to `global` object (write is side effect)
local non-const read/write	get/set value of field
C built-in operators $\oplus$	return result of performing $\oplus$
invocation of function `f`	await side effects, create `C_f` object, make synchronous call to method `call` of object
Unspecified evaluation order	Asynchronous method calls to **this** object
Sequence points	**await** statements

Table 1: Translation of C concepts into ABS

call objects each (sub)expression being evaluated is modeled as an asynchronous method call to itself with **await** statements modeling *sequence points*: the point between evaluation of all arguments and side effects of a function call, and the call itself; the semicolon at the end of an expression statement; etc. Access to global variables is modeled by methods making blocking calls to the `global` object, while (potentially recursive) function calls are modeled by creating new Active Objects for the appropriate function and making blocking calls to these new objects.

Example 3. *Consider the function* **main** *in Fig. 1 and the statement* **return** `x + id_set_x(1);` *inside, where there is a sequence point between evaluation of the expression and returning from the function. The ABS class extracted is shown in Fig. 5, where the method* `call` *models function execution and lines 5-9 model the unspecified evaluation order of the the expression* `x + id_set_x(1)` *with the* **await** *at line 10 allowing non-deterministic choice in which order the methods to* **this** *are executed in. Once all futures have been resolved, the* **await** *regains control, modeling the sequence point before returning. The method* `call` *then returns the value of the addition. The method* `get_global_x` *models the memory access, by making a synchronous call*[5] *to the* `global` *parameter of the class, requesting the value of* `x`*. The method* `call_id_set_x_val_0` *models a call to the function* `id_set_x` *with an argument evaluated at compile time and zero side effects from evaluating its argument*[6]*. This is done by first creating a new* `C_id_set_x` *object with access to the same* `global` *object and then making a synchronous call to the* `call` *method of that object with the evaluated function arguments as parameters. Finally, the method* `op_plus_fut_fut` *models the addition of two subexpressions evaluated at runtime and therefore modeled as futures. First, the method awaits the resolution of its subexpressions, then returns the sum. While the three methods can be executed in arbitrary (and interleaving) order, the only visible difference depends on the order of* `get_global_x` *and* `call_id_set_x_val_0`*, as* `op_plus_fut_fut` *immediately awaits resolution of the*

[5] An asynchronous call to an object in another object immediately followed by a **get**.

[6] If the argument were a future or side effects (modeled as futures) were present, the method would immediately await resolution of all these futures.

```
class C_main(Global global)
      implements I_main {
 Int call() {
  ...
  Fut<Int> fut_x = this!get_global_x();
  Fut<Int> fut_set =
    this!call_id_set_x_val_0(1);
  Fut<Int> fut_add =
    this!op_plus_fut_fut(fut_x, fut_set);
  await fut_x? & fut_set? & fut_add;
  return fut_add.get;
 }
 Int get_global_x() {
  Fut<Int> f = global!get_x();
  return f.get;
 }
 Int call_id_set_x_val_0(Int arg1) {
  I_id_set_x o = new C_id_set_x(global);
  Fut<Int> f = o!call(arg1); return f.get;
 }
 Int op_plus_fut_fut(Fut<Int> fut_arg1,
                     Fut<Int> fut_arg2) {
  await fut_arg1? & fut_arg2?;
  Int arg1 = fut_arg1.get;
  Int arg2 = fut_arg2.get;
  return arg1 + arg2;
 }
}
```

Figure 5: Class `C_main` extracted from function **main** in Fig. 1

other two methods.

4.2 Automatically Synthesizing Specifications

Due to the automated nature in which function-modelling classes and helper methods are generated, we can synthesize some specifications directly. For others we require ACSL specification of the underlying C program.

Auto-generate specifications related to global object As each function-modelling class receives the `global` object as a parameter, uses it to access global variables and passes it on when instantiating any further function-modelling classes, we must (at least) specify that this class parameter (and field) is never **null**. To this end all function-modelling classes are specified with:

```
[Spec : Requires(global != null)]
[Spec : ObjInv(global != null)]
```

Auto-generate precise postconditions for operator methods C2ABS-generated methods from C built-in operators $\oplus$ all perform the same basic steps: await resolution of all future parameters and then return the result of performing $\oplus$ on the (resolved) parameters. Precise postcondition specifications for each of these methods can therefore be generated automatically, by ensuring that the result of the method is equal to the result of performing $\oplus$ on the (resolved) parameters. All C operator method declarations in interfaces are thus automatically annotated with appropriate postcondition specifications.

Example 4. *The interface* `I_main` *in the model extracted from the program in Fig. 1 contains the following annotated method declaration:*

```
[Spec : Ensures(valueof(fut_arg1) + valueof(fut_arg2) == result)]
Int op_plus_fut_fut(Fut<Int> fut_arg1,
                    Fut<Int> fut_arg2);
```

Translate ACSL requires/ensures function contracts ACSL requires/ensures clauses specify (relational) restrictions upon the function arguments and functional guarantees for the result. Following similar steps to those for extracting C expressions—simplified somewhat due to lack of side effects—these can be converted into pre- and postconditions of the `call` method in the interface modelling the function. Additionally, similar pre- and postconditions are added to the indirect call methods in any interfaces modelling functions calling the specified function. When an argument to an indirect call is a future value, the pre- and postconditions must be formulated to hold for the resolved argument.

Example 5. *Given the specified function* `id_set_x` *at line 3 in Fig. 2:*

```
2 int id_set_x(int val)
3 /*@ requires val == 1; ensures \result == 1; @*/ {
```

We annotate both the `call` *method in* `I_id_set_x` *and the* `call_id_set_x_val` *method in* `I_main` *with the following specifications:*

```
[Spec : Requires(val == 1)]
[Spec : Ensures(result == 1)]
```

Translate ACSL strong global invariants While a strong global invariant must hold at every point in the program, it suffices to *check* that it holds at program start and whenever the global variable is changed. The ACSL invariant is translated as above and added as an object invariant in the `Global` class and as preconditions on the argument of all setter methods for said variable. When the argument to indirect setters outside of `Global` is a future value, the precondition must be formulated to hold for the resolved argument. In order to *use* the invariant, we add postconditions to all getter methods for the variable.

Example 6. *Given the strong global invariant at line 1 in Fig. 2 that* `x == 0 || x == 1`*, the global state is modeled as the code in Fig. 6. Additionally,* `I_id_set_x` *and* `I_main` *contain the annotated method declarations in the lower code in Fig. 6.*

Use ABS functions in lieu of ACSL logic functions ACSL allows pure *logic functions* to be defined (inductively or axiomatically) and called in ACSL specifications. Translating these definitions is outside of the scope of this work and we therefore instead allow ABS functions to be called directly in ACSL specifications. If the ABS function is not inside the standard library, it must be defined inside an ACSL-style comment in the C program.

Scope The C Standard lists 52 cases of unspecified behavior [23, Annex. J.1]. However, most of these cases are not relevant to functional verification of runtime semantics, e.g., unspecified behavior of macros; or concern well-investigated elements outside of the considered language fragment, such as

```
interface Global {
 [Spec : Ensures(result == 0||result==1)]
 Int get_x();
 [Spec : Requires(arg == 0||arg == 1)]
 Unit set_x(Int arg);
}
[Spec : ObjInv(this.x == 0 || this.x == 1)]
class Global implements Global {
 Int x = 0;
 Int get_x() { return this.x; }
 Unit set_x(Int arg) {
  this.x = arg;
  return unit;
 }
}
```

```
[Spec:Requires(arg == 0 || arg == 1)]
Unit set_global_x_val(Int arg);
[Spec:Requires(valueof(fut_arg) == 0||valueof(fut_arg) == 1)]
Unit set_global_x_fut(Fut<Int> fut_arg);
[Spec:Ensures(result == 0||result == 1)]
Int get_global_x();
```

Figure 6: Example for translating strong global invariants.

floating points and string literals; or concern deprecated features of old compilers for rare hardware, such as the use of negative zeros in integer types. Our focus is therefore on those cases that touch on core aspects of the runtime semantics and are relevant for almost all programs: order of subexpression and side effect evaluation (except for some operators such as &&) [23, 6.5], of function argument evaluation [23, 6.5.2.2] and of evaluation of complex assignments [23, 6.5.16]. All these aspects can be handled by our approach and reduced to non-determinism of concurrent systems.

5 Case Study

Underspecified behavior lurks at almost every binary operation[7] and can have subtle effects in larger systems. To evaluate our verification approach, we use an extreme case of underspecification, investigating the C program[8] in Fig. 7 containing a function whose result heavily depends on unspecified evaluation order. The function in question is declared as `int one_to_fib(int n)` and should calculate a number between 1 and the nth Fibonacci number. The base cases are for inputs 1 and 2 (as well as all non-positive inputs), which return 1; as well as for input 3, which returns either 1 or 2 in the same manner as the program in Figure 1. Otherwise, `one_to_fib(n)` returns the sum of `one_to_fib(n-2)` and `one_to_fib(n-1)` with a potential decrement of 1 in the function `pred_or_id` ensuring that 1 is always a potential result, as otherwise $\{1,\dots,Fib(n-1)\}+\{1,\dots,Fib(n-2)\} = \{2,\dots,Fib(n)\}$.

Verification of this program is a challenging task due to the extensive non-determinism. In [37] the extracted model for this program was exhaustively checked for inputs up to 5, validating that all possible outputs (and no outputs outside this range) could be produced. Later experiments with an enhanced

[7]Underspecified behavior also lurks at many function calls.

[8]Adapted from an idea on Derek Jones's *The Shape of Code* blog at:
`shape-of-code.coding-guidelines.com/2011/06/18/fibonacci-and-jit-compilers/`

```
 1 //@ ABS def Int fib(Int n) = if n <= 2 then 1
 2 //@                         else fib(n−1) + fib(n−2);
 3
 4 /*@ strong global invariant x == 0 || x == 1; @*/ int x;
 5
 6 //@ ensures \result == val;
 7 int id_set_x(const int val)
 8 { x=1; return val; }
 9 //@ ensures \result == 1 || \result == 2;
10 int one_or_two(void) {
11     x=0;
12     return x + id_set_x(1);
13 }
14 //@ ensures \result == val − 1 || \result == val;
15 int pred_or_id(const int val) {
16     x=0;
17     return val - x + id_set_x(0);
18 }
19 //@ ensures \result >= 1 && \result <= fib(n);
20 int one_to_fib(const int n) {
21  if (n > 3)
22   return one_to_fib(n-2)
23         + pred_or_id(one_to_fib(n-1));
24  else if (n == 3) return one_or_two();
25  else return 1; }
```

Figure 7: Calculate a number between 1 and the `n`th Fibonacci number in C

model extraction process partially validated models for inputs up to 10. In this work we verify that no outputs outside of the range are produced for any (valid) inputs.[9] The annotated extracted model for this C program can be found in the technical report. The ABS function definition inside the ACSL-style specification in line 2 is copied verbatim into the model, the helper methods for `+` (used in lines 10, 15 and 23) and `-` (line 15) receive precise specifications, the strong global invariant on `x` at line 4 produces specifications throughout the model (`Global` interface and class, plus indirect getter and setter methods of other interfaces), while the `call` methods and their indirect callers are specified with translations of the contracts for the matching functions. As the program does not contain a **main** method and is not executable, so the model it produces is therefore also not executable: the main block in the extracted model is empty. As we are focused on proving a property of `one_to_fib` in general, rather than for a specific actual call, this non-executability is not a problem. This shows an additional strength of our approach, in that we can analyze *library* calls in isolation, rather than only being able to analyze a complete program. `Crowbar` can close all proof obligations of the extracted model *automatically*. Note that we prove the following *for all inputs* to `one_to_fib`.

Theorem 1. *The extracted model is safe with respect to its specification.*

In particular, the proof cannot be closed if we change the specification to express that `one_to_fib` returns a value from a smaller range.

[9]The semantics of the program are underspecified but *not* undefined.

Deadlock Freedom. Running `Crowbar` performs a simple analysis for structurally deadlock-free methods and returns all methods for which it cannot deduce it. For the extracted model it returns 9 such methods. These are the methods that take futures as parameters, which is not supported by the deadlock analysis in `Crowbar`, and methods depending on these methods. However, all futures that are passed as parameters are always futures of free methods. Thus we can state the following lemma, which is proven in the technical report.

Lemma 1. *The extracted model is deadlock free for every extractable main block.*

Applying State-of-the-Art Tools. As detailed in Sec. 1, other automatic tools cannot handle the example correctly. They either fix an evaluation order and can (wrongly) prove a stronger result, i.e., that the result is always the *n*th Fibonacci number (Frama-C, RV-match), do not support specification of global invariants of ACSL (Frama-C) or do not support verification at all(Cerberus). We do not compare our approach explicitly with the theory presented by Frumin et al. [17], which does treat underspecification correctly, but not for C and requires manual translation and manual specification of the translated program in the target formalism and an interactive proof.

6 Conclusion

We have demonstrated a novel approach combining model extraction with deductive verification of a distributed active objects model in order to verify C programs with underspecified behavior by reducing the non-determinism of underspecification to non-determinism of parallelism. We have extended the `C2ABS` tool—which already gives C a formal semantics in terms of Active Objects— to automatically translate a large subset of ACSL specifications into BPL specifications and implemented the `Crowbar` tool based on [26] in order to verify the specified model and analyze it for deadlock freedom. Using a complex case study that exemplifies the challenges for verification of underspecified programs we showed that our approach of model extraction and verification is *fully automatic*. We reused a standard logic and deadlock analysis for ABS and did not need special amendments for underspecified behavior after the extraction.

Future Work. For formalized parallelization of C code, we plan to integrate a formal, logic-based dependences analysis [9] and to consider further cases of underspecification of a larger fragment of C, e.g., in list initializers. The newest version of `C2ABS` uses different model extraction strategies [38] and we will investigate using `Crowbar` to verify these models as well. In cases where the input C program is not completely specified, we envisage generating the missing object invariants and method contracts automatically via counter-example guided refinement techniques [11] using the failed `Crowbar` proofs.

Finally, it is worth investigating how our model extraction approach compares to an explicit handling of underspecifiation by branching for every possible evaluation order.

References

[1] `https://runtimeverification.com/match/`.

[2] `https://github.com/kframework/c-semantics`.

[3] `http://kframework.org/`.

[4] https://cerberus.cl.cam.ac.uk/.

[5] https://coq.inria.fr/.

[6] BAKER, H. G., AND HEWITT, C. E. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages* (August 1977), no. 12 in SIGPLAN Notices, p. 11.

[7] BARRETT, C. W., STUMP, A., AND TINELLI, C. The smt-lib standard version 2.0.

[8] BAUDIN, P., CUOQ, P., FILLIÂTRE, J.-C., MARCHÉ, C., MONATE, B., MOY, Y., AND PREVOSTO, V. ACSL: ANSI/ISO C specification language version 1.14, 2018. https://frama-c.com/acsl.html.

[9] BUBEL, R., HÄHNLE, R., AND HEYDARI TABAR, A. A program logic for dependence analysis. In *IFM* (2019), vol. 11918 of *LNCS*.

[10] BÜHLER, D., CUOQ, P., AND YAKOBOWSKI, B. Eva – the evolved value analysis plug-in, manual v21.1, 2020.

[11] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement. In *Computer Aided Verification* (Berlin, Heidelberg, 2000), E. A. Emerson and A. P. Sistla, Eds., Springer Berlin Heidelberg, pp. 154–169.

[12] CUOQ, P., KIRCHNER, F., KOSMATOV, N., PREVOSTO, V., SIGNOLES, J., AND YAKOBOWSKI, B. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (Berlin, Heidelberg, 2012), SEFM'12, Springer-Verlag, p. 233–247.

[13] DE BOER, F. S., SERBANESCU, V., HÄHNLE, R., HENRIO, L., ROCHAS, J., DIN, C. C., JOHNSEN, E. B., SIRJANI, M., KHAMESPANAH, E., FERNANDEZ-REYES, K., AND YANG, A. M. A survey of active object languages. *ACM Comput. Surv. 50*, 5 (2017), 76:1–76:39.

[14] DIN, C. C., BUBEL, R., HÄHNLE, R., GIACHINO, E., LANEVE, C., AND LIENHARDT, M. Deliverable D3.2 Verification of project FP7-610582 (ENVISAGE), Mar. 2015. available at http://www.envisage-project.eu.

[15] DIN, C. C., AND OWE, O. Compositional reasoning about active objects with shared futures. *Formal Asp. Comput. 27*, 3 (2015), 551–572.

[16] ELLISON, C., AND ROSU, G. An executable formal semantics of C with applications. In *POPL'12* (2012), ACM, pp. 533–544.

[17] FRUMIN, D., GONDELMAN, L., AND KREBBERS, R. Semi-automated reasoning about non-determinism in C expressions. In *ESOP* (2019), vol. 11423 of *LNCS*.

[18] HÄHNLE, R., HEYDARI TABAR, A., MAZAHERI, A., NOROUZI, M., STEINHÖFEL, D., AND WOLF, F. Safer parallelization. In *ISoLA (2)* (2020), vol. 12477 of *Lecture Notes in Computer Science*, Springer, pp. 117–137.

[19] HATHHORN, C., ELLISON, C., AND ROŞU, G. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (June 2015), ACM, pp. 336–345.

[20] HENRIO, L., LANEVE, C., AND MASTANDREA, V. Analysis of synchronisations in stateful active objects. In *Integrated Formal Methods* (Cham, 2017), N. Polikarpova and S. Schneider, Eds., Springer International Publishing, pp. 195–210.

[21] HEWITT, C., BISHOP, P., AND STEIGER, R. A universal modular ACTOR formalism for artificial intelligence. IJCAI'73, Morgan Kaufmann Publishers Inc.

[22] HOLZMANN, G. J., AND SMITH, M. H. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Software Eng. 28*, 4 (2002), 364–377.

[23] ISO. ISO C standard 1999. Tech. rep., 1999. ISO/IEC 9899:1999 draft.

[24] JOHNSEN, E. B., HÄHNLE, R., SCHÄFER, J., SCHLATTE, R., AND STEFFEN, M. ABS: A core language for abstract behavioral specification. In *FMCO 2010* (2010), B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds., vol. 6957 of *LNCS*, Springer, pp. 142–164.

[25] Kamburjan, E. Detecting deadlocks in formal system models with condition synchronization. *ECEASST 76* (2018).

[26] Kamburjan, E. Behavioral program logic. In *TABLEAUX* (2019), vol. 11714 of *LNCS*, Springer, pp. 391–408.

[27] Kamburjan, E., Din, C. C., Hähnle, R., and Johnsen, E. B. Behavioral contracts for cooperative scheduling, 2020.

[28] Kamburjan, E., and Hähnle, R. Prototyping formal system models with active objects. In *ICE* (2018), vol. 279 of *EPTCS*, pp. 52–67.

[29] Kamburjan, E., Scaletta, M., and Rollshausen, N. Crowbar: Behavioral symbolic execution for deductive verification of active objects. *CoRR abs/2102.10127* (2021).

[30] Krebbers, R. An operational and axiomatic semantics for non-determinism and sequence points in C. In *POPL'14* (2014), ACM.

[31] Lebesbye, T., Mauro, J., Turin, G., and Yu, I. C. Boreas - A service scheduler for optimal kubernetes deployment. In *ICSOC* (2021), vol. 13121 of *Lecture Notes in Computer Science*, Springer, pp. 221–237.

[32] Lin, J., Lee, M., Yu, I. C., and Johnsen, E. B. A configurable and executable model of spark streaming on apache yarn. *International Journal of Grid and Utility Computing 11*, 2 (2020), 185–195.

[33] Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R. N. M., and Sewell, P. Into the depths of C: elaborating the de facto standards. In *37th PLDI* (2016), ACM.

[34] Norrish, M. C formalised in HOL. Tech. Rep. UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, Dec. 1998.

[35] Papaspyrou, N. S. Denotational semantics of ANSI C. *Computer Standards & Interfaces 23*, 3 (2001), 169–185.

[36] Roşu, G., and Şerbănuţă, T. F. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming 79*, 6 (2010), 397–434.

[37] Wasser, N., Heydari Tabar, A., and Hähnle, R. Modeling non-deterministic C code with active objects. In *FSEN* (2019), vol. 11761 of *LNCS*.

[38] Wasser, N., Heydari Tabar, A., and Hähnle, R. Automated model extraction: From non-deterministic c code to active objects. *Science of Computer Programming 204* (2021), 102597.

Acknowledgements

This work was partially funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project and partially by the Research Council of Norway via `PeTWIN` (Grant Nr. 294600) and `SIRIUS` (Grant Nr. 237898).

Demystifying Attestation in Intel Trust Domain Extensions (TDX) via Formal Verification

Muhammad Usama Sardar and Christof Fetzer

Intel Trust Domain Extensions (TDX) is the next-generation confidential computing offering of Intel. One of the most critical processes of Intel TDX is the remote attestation mechanism. Since remote attestation bootstraps trust in remote applications, any vulnerability in the attestation mechanism can therefore impact the security of an application. Hence, we investigate the use of formal methods to ensure the correctness of the attestation mechanisms. The symbolic security analysis of remote attestation protocol in Intel TDX reveals a number of subtle inconsistencies found in the specification of Intel TDX that could potentially lead to design and implementation errors as well as attacks. These inconsistencies have been reported to Intel and Intel is in process of updating the specifications. We also explain how formal specification and verification using ProVerif could help avoid these flaws and attacks.

Session Fidelity for ElixirST: A Session-Based Type System for Elixir Modules

Gerard Tabone Adrian Francalanza
Computer Science Department
University of Malta
Msida, Malta
gerard.tabone.17@um.edu.mt adrian.francalanza@um.edu.mt

This paper builds on prior work investigating the adaptation of session types to provide behavioural information about Elixir modules. A type system called ElixirST has been constructed to statically determine whether functions in an Elixir module observe their endpoint specifications, expressed as session types; a corresponding tool automating this typechecking has also been constructed. In this paper we formally validate this type system. An LTS-based operational semantics for the language fragment supported by the type system is developed, modelling its runtime behaviour when invoked by the module client. This operational semantics is then used to prove session fidelity for ElixirST.

1 Introduction

In order to better utilise recent advances in microprocessor design and architecture distribution, modern programming languages offer a variety of abstractions for the construction of concurrent programs. In the case of message-passing programs, concurrency manifests itself as spawned computation that exhibits *communication as a side-effect*, potentially influencing the execution of other (concurrent) computation. Such side-effects inevitably increase the complexity of the programs produced and lead to new sources of errors. As a consequence, program correctness becomes harder to verify and language support for detecting errors at the development stage can substantially decrease the number of concurrency errors.

Elixir [34], based on the actor model [1, 14], is one such example of a modern programming language for concurrency. As depicted in Figure 1, Elixir programs are structured as a collection of *modules* that contain *functions*, the basic unit of code decomposition in the language. A module only exposes a subset of these functions to external invocations by defining them as *public*; these functions act as the only entry points to the functionality encapsulated by a module. Internally, the bodies of these public functions may then invoke other functions, which can either be the *public* ones already exposed or the *private* functions that can only be invoked from within the same module. For instance, Figure 1 depicts a module m which contains several public functions (*i.e.,* $f_1, \ldots, f_n$) and private functions (*i.e.,* $g_1, \ldots, g_j$). For example, the public function f_1 delegates part of its computation by calling the private functions g_1 and g_j, whereas the body of the public function f_n invokes the other public function f_1 when executed. Internally, the body of the private function g_1 calls the other private function g_2 whereas the private function g_j can recursively call itself.

A prevalent Elixir design pattern is that of a server listening for client requests. For each request, the server spawns a (public) function to execute independently and act as a dedicated client handler: after the respective process IDs of the client and the spawned handler are made known to each other, a session of interaction commences between the two concurrent entities (via message-passing). For instance, in Figure 1, a handler process running public function f_1 is assigned to the session with client client_1 whereas

Preliminary Report. Final version to appear in:
ICE 2022

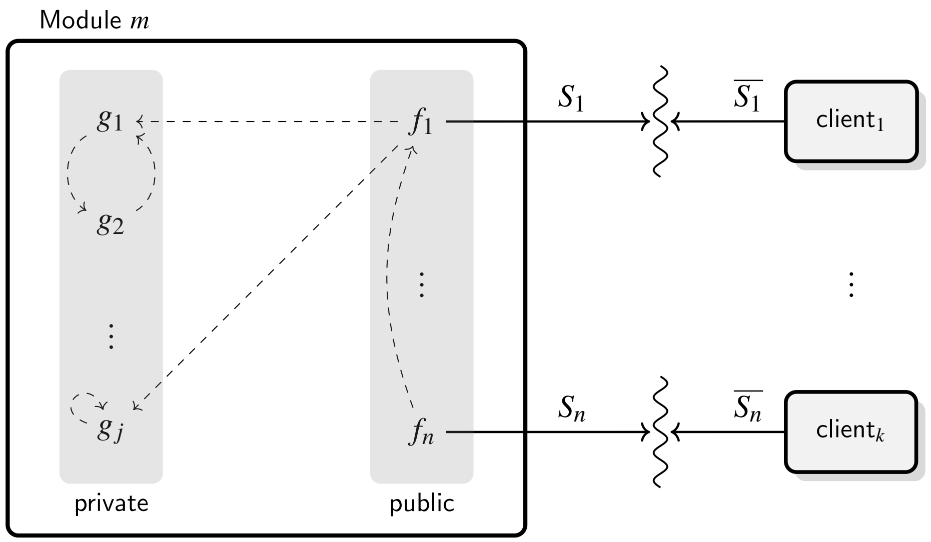


Figure 1: An Elixir module consisting of public and private functions, interacting with client processes

the request from client$_k$ is assigned a dedicated handler running function f_n. Although traditional interface elements such as function parameters (used to instantiate the executing function body with values such as the client process ID) and the function return value (reporting the eventual outcome of handled request) are important, the messages exchanged between the two concurrent parties within a session are equally important for software correctness. More specifically, communication incompatibilities between the interacting parties could lead to various runtime errors. For example, if in a session a message is sent with an unexpected payload, it could cause the receiver's subsequent computation depending on it to crash (*e.g.* multiplying by a string when a number should have been received instead). Also, if messages are exchanged in an incorrect order, they may cause *deadlocks* (*e.g.* two processes waiting forever for one another to send messages of a particular kind when a message of a different kind has been sent instead).

In many cases, the expected protocol of interactions within a session can be statically determined from the respective endpoint implementations, namely the function bodies; for simplicity, our discussion assumes that endpoint interaction protocols are dual, *e.g.* S_1 and $\overline{S_1}$ in Figure 1. Although Elixir provides mechanisms for specifying (and checking) the parameters and return values of a function within a module, it does *not* provide any support for describing (and verifying) the interaction protocol of a function in terms of its communication side-effects. To this end, in earlier work [32] we devised the tool[1] ElixirST, assisting module construction in two ways: (a) it allows module designers to formalise the session endpoint protocol as a session type, and ascribe it to a public function; (b) it implements a type-checker that verifies whether the body of a function respects the ascribed session type protocol specification.

Contribution. This paper validates the underlying type system on which the ElixirST type-checker is built. More concretely, in Section 3 we formalise the runtime semantics of the Elixir language fragment supported by ElixirST as a labelled transition system (LTS), modelling the execution of a spawned handler interacting with a client within a session. This operational semantics then allows us to prove *session fidelity* for the ElixirST type system in Section 4. In Section 2 we provide the necessary background on the existing session type system from [32] to make the paper self-contained.

[1] ElixirST is available on GitHub: `https://github.com/gertab/ElixirST`

2 Preliminaries

We introduce a core Elixir subset and review the main typing rules for the ElixirST type system [32].

2.1 The Actor Model

Elixir uses the actor concurrency model [1, 14]. It describes computation as a group of concurrent processes, called *actors*, which do *not* share any memory and interact exclusively via asynchronous messages. Every actor is identified via a unique process identifier (*pid*) which is used as the address when sending messages to a specific actor. Messages are communicated asynchronously, and stored in the mailbox of the addressee actor. An actor is the only entity that can fetch messages from its mailbox, using mechanisms such as pattern matching. Apart from sending and reading messages, an actor can also spawn other actors and obtain their fresh *pid* as a result; this *pid* can be communicated as a value to other actors via messaging.

2.2 Session Types

The ElixirST type system [32] assumes the standard expression types, including basic types, such as boolean, number, atom and pid, and inductively defined types, such as tuples ($\{T_1, \ldots, T_n\}$) and lists ($[T]$); these already exist in the Elixir language and they are dynamically checked. It extends these with (binary) session types, which are used to statically check the message-passing interactions.

$$\text{Expression types} \qquad T ::= \mathsf{boolean} \mid \mathsf{number} \mid \mathsf{atom} \mid \mathsf{pid} \mid \{T_1, \ldots, T_n\} \mid [T]$$

$$\begin{array}{llll} \text{Session types} \quad S ::= \&\big\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\big\}_{i\in I} & \text{Branch} & \mid \mathsf{rec}\,\mathsf{X}.S & \text{Recursion} \\ \qquad\qquad\qquad \mid \oplus\big\{!\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\big\}_{i\in I} & \text{Choice} & \mid \mathsf{X} & \text{Variable} \\ \qquad\qquad\qquad \mid \mathsf{end} & \text{Termination} & & \end{array}$$

The *branching* construct, $\&\big\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\big\}_{i\in I}$, requires the code to be able to receive a message that is labelled by any one of the labels $\mathtt{l}_i$, with the respective list of values of type $\widetilde{T_i}$ (where $\tilde{T}$ stands for $T^1,\ldots,T^k$ for some $k \geq 0$), and then adhere to the continuation session type S_i. The *choice* construct is its dual and describes the range and format of outputs the code is allowed to perform. In both cases, the labels need to be pairwise distinct. Recursive types are treated equi-recursively [27], and used interchangeably with their unfolded counterparts. For brevity, the symbols & and $\oplus$ are occasionally omitted for singleton options, *e.g.*, $\oplus\big\{!\mathtt{l}(\mathsf{number}).S_1\big\}$ is written as $!\mathtt{l}(\mathsf{number}).S_1$; similarly end may be omitted as well, *e.g.*, $?\mathtt{l}()$ stands for $?\mathtt{l}().\mathsf{end}$. The *dual* of a session type S is denoted as $\overline{S}$.

2.3 Elixir Syntax

Elixir programs are organised as modules, *i.e.,* `defmodule` m `do` $\widetilde{P}\,\widetilde{D}$ `end`. Modules are defined by their name, m, and contain two sets of public $\widetilde{D}$ and private $\widetilde{P}$ functions, declared as sequences. Public functions, `def` $f(y,\widetilde{x})$ `do` t `end`, are defined by the **`def`** keyword, and can be called from any module. In contrast, private functions, `defp` $f(y,\widetilde{x})$ `do` t `end`, can only be called from within the defining module. Functions are defined by their name, f, and their body, t, and parametrised by a sequence of *distinct* variables, $y,\widetilde{x}$, the length of which, $|y,\widetilde{x}|$, is called the *arity*. As an extension to [32], the first parameter

$$\begin{array}{rrcl}
\text{Module} & M & ::= & \texttt{defmodule}\ m\ \texttt{do}\ \widetilde{P}\ \widetilde{D}\ \texttt{end} \\
\text{Public fun.} & D & ::= & K\quad B\quad \texttt{def}\ f(y,\ \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \\
\text{Private fun.} & P & ::= & B\quad \texttt{defp}\ f(y,\ \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \\
\text{Type ann.} & B & ::= & \texttt{@spec}\ f\big(\widetilde{T}\big)\ ::\ T \\
\text{Session ann.} & K & ::= & \texttt{@session}\ \text{“}\mathsf{X} = S\text{”} \\
 & & \mid & \texttt{@dual}\ \text{“}\mathsf{X}\text{”} \\
 & & & \\
\text{Expressions} & e & ::= & w \\
 & & \mid & \texttt{not}\ e \mid e_1 \diamond e_2 \\
 & & \mid & [e_1 \mid e_2] \mid \{e_1,\ \ldots,\ e_n\} \\
\text{Operators} & \diamond & ::= & < \mid > \mid <= \mid >= \mid == \\
 & & \mid & != \mid + \mid - \mid * \mid / \mid \texttt{and} \mid \texttt{or}
\end{array}$$

$$\begin{array}{rrcl}
\text{Basic val.} & b & ::= & \mathit{boolean} \mid \mathit{number} \mid \mathit{atom} \mid \mathit{pid} \\
 & & \mid & [\,] \\
\text{Values} & v & ::= & b \mid [v_1 \mid v_2] \mid \{v_1,\ \ldots,\ v_n\} \\
\text{Identifiers} & w & ::= & b \mid x \\
\text{Patterns} & p & ::= & w \mid [w_1 \mid w_2] \mid \{w_1,\ \ldots,\ w_n\} \\
\text{Terms} & t & ::= & e \\
 & & \mid & x = t_1;\ t_2 \\
 & & \mid & \texttt{send}\,(w, \{\,\texttt{:l}, e_1,\ \ldots,\ e_n\}) \\
 & & \mid & \texttt{receive do} \\
 & & & \quad \big(\{\,\texttt{:l}_i, p_i^1,\ \ldots,\ p_i^n\} \to t_i\big)_{i\in I}\texttt{end} \\
 & & \mid & f\,(w,\ e_1,\ \ldots,\ e_n) \\
 & & \mid & \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}
\end{array}$$

Figure 2: Elixir syntax

(y), is reserved for the *pid* of the dual process. Although a module may contain functions with the same name, their arity must be different.

In our formalisation, Elixir function parameters and return values are assigned a type using the `@spec` annotation, $f(\widetilde{T}) :: T$, describing the parameter types, $\widetilde{T}$, and the return type, T. This annotation is already used by Dialyzer for success typing [21]. In addition to this, we decorate public functions with session types, defined in Section 2.2, to describe their side-effect protocol. Public functions can be annotated directly using `@session` “$\mathsf{X} = S$”, or indirectly using the dual session type, `@dual` “X”, where $\mathsf{X} = S$ is shorthand for $\mathsf{rec}\,\mathsf{X}\,.\,S$.

The body of a function consists of a term, t, which can take the form of an expression, a `let` statement, a send or receive construct, a case statement or a function call; see Figure 2. In the case of the `let` construct, $x = t_1;\ t_2$, the variable x is a *binder* for the variables in t_2 acting as a placeholder for the value that the subterm t_1 evaluates to. We write $t_1; t_2$, as *syntactic sugar* for $x = t_1;\ t_2$ whenever x is not used in t_2. The *send* statement, $\texttt{send}\,(x, \{\,\texttt{:l}, e_1,\ \ldots,\ e_n\})$, allows a process to send a message to the *pid* stored in the variable x, containing a message $\{\,\texttt{:l}, e_1,\ \ldots,\ e_n\}$, where `:l` is the label. The *receive* construct, $\texttt{receive do}\ \big(\{\,\texttt{:l}_i, p_i^1,\ \ldots,\ p_i^n\} \to t_i\big)_{i\in I}\texttt{end}$, allows a process to receive a message tagged with a label that matches one of the labels $\texttt{:l}_i$ and a list of payloads that match the patterns $p_i^1,\ \ldots,\ p_i^n$, branching to continue executing as t_i. Patterns, p, defined in Figure 2, can take the form of a variable, a basic value, a tuple or a list (*e.g.* $[x \mid y]$, where x is the head and y is the tail of the list). The remaining constructs are fairly standard. Variables in patterns $p_i^1,\ \ldots,\ p_i^n$ employed by the `receive` and `case` statements are binders for the respective continuation branches t_i. We assume standard notions of open (*i.e.,* $\mathbf{fv}(t) \neq \emptyset$) and closed (*i.e.,* $\mathbf{fv}(t) = \emptyset$) terms and work up to alpha-conversion of bound variables.

2.4 Type System

The session type system from [32] statically verifies that public functions within a module observe the communication protocols ascribed to them. It uses three environments:

$$\begin{array}{rl}
\text{Variable binding env.} & \Gamma ::= \emptyset \mid \Gamma,\ x:T \\
\text{Session typing env.} & \Delta ::= \emptyset \mid \Delta,\ f/n:S \\
\text{Function inf. env.} & \Sigma ::= \emptyset \mid \Sigma,\ f/n : \left\{\begin{array}{l}\texttt{params} = \widetilde{x},\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{body} = t,\ \texttt{return\_type} = T,\ \texttt{dual} = y\end{array}\right\}
\end{array}$$

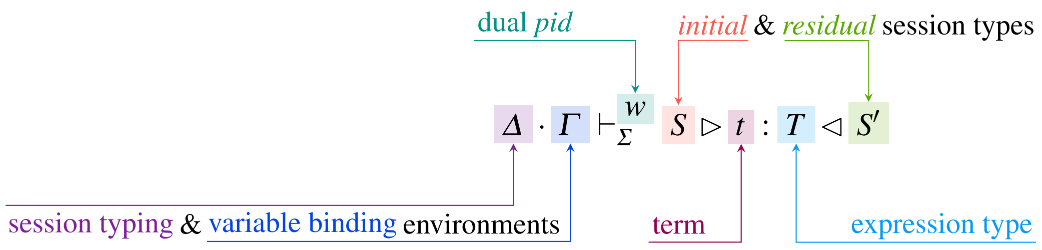
The *variable binding* environment, Γ, maps (data) variables to basic types ($x:T$). We write $\Gamma, x:T$ to extend Γ with the new mapping $x:T$, where $x \notin \mathbf{dom}(\Gamma)$. The *session typing* environment, Δ, maps function names and arity pairs to their session type ($f/n:S$). If a function f/n has a *known* session type, then it can be found in Δ, *i.e.,* $\Delta(f/n) = S$. Each module has a static *function information* environment, Σ, that holds information related to the function definitions. For a function f, with arity n, $\Sigma(f/n)$ returns the tail list of parameters (`params`) and their types (`param_types`), the function's body (`body`), and its return type (`return_type`). In contrast to the original type system from [32], $\Sigma(f/n)$ also returns the variable name that represents the interacting process' *pid*, *i.e.,* the option `dual`. We assume that *function information* environments, Σ, are *well-formed*, meaning that all functions mapped ($f/n \in \mathbf{dom}(\Sigma)$) observe the following condition requiring that the body of function f/n is *closed*:

$$\mathbf{fv}\big(\Sigma(f/n).\texttt{body}\big) \setminus \big(\Sigma(f/n).\texttt{params} \cup \Sigma(f/n).\texttt{dual}\big) = \emptyset$$

Session typechecking is initiated by analysing an Elixir module, rule [tModule]. A module is typechecked by inspecting each of its public functions, ascertaining that they correspond and fully consume the session types ascribed to them. The rule uses three helper functions. The function $\mathbf{functions}(\widetilde{D})$ returns a list of all function names (and arity) of the public functions ($\widetilde{D}$) to be checked individually. The function $\mathbf{sessions}(\widetilde{D})$ obtains a mapping of all the public functions to their expected session types stored in Δ. This ensures that when a function f with arity n executes, it adheres to the session type associated with it using either the `@session` or `@dual` annotations. The helper function **details** populates the *function information* environment (Σ) with details about all the *public* ($\widetilde{D}$) and *private* functions ($\widetilde{P}$) within the module.

$$[\text{tModule}]\ \frac{\begin{array}{c}\Delta = \mathbf{sessions}(\widetilde{D}) \qquad \Sigma = \mathbf{details}(\widetilde{P}\,\widetilde{D}) \\ \forall f/n \in \mathbf{functions}(\widetilde{D}) \cdot \left\{\begin{array}{l}\Delta(f/n) = S \qquad \Sigma\,(f/n) = \Omega \\ \Omega\,.\texttt{params} = \widetilde{x} \qquad \Omega\,.\texttt{param\_types} = \widetilde{T} \\ \Omega.\texttt{body} = t \qquad \Omega.\texttt{return\_type} = T \qquad \Omega.\texttt{dual} = y \\ \Delta \cdot \big(y:\mathsf{pid}, \widetilde{x}:\widetilde{T}\big) \vdash^{y}_{\Sigma} S \rhd t : T \lhd \mathsf{end}\end{array}\right.\end{array}}{\vdash \texttt{defmodule}\ m\ \texttt{do}\ \widetilde{P}\ \widetilde{D}\ \texttt{end}}$$

For every public function f/n in $\mathbf{functions}(\widetilde{D})$, [tModule] checks that its body adheres to it session type using the highlighted *term typing* judgement detailed below:


dual *pid*
initial & *residual* session types
$\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$
session typing & variable binding environments
term
expression type

This judgement states that "the term t can produce a value of type T after following an interaction protocol starting from the initial session type S up to the residual session type S', while interacting with a dual process with *pid* identifier w. This typing is valid under some *session typing* environment Δ, *variable binding* environment Γ and *function information* environment Σ." Since the *function information* environment Σ is static for the whole module (and by extension, for all sub-terms), it is left implicit in the term typing rules. We consider four main rules, and relegate the rest to Appendix B.1.

$$[\textsc{tBranch}]\ \frac{\forall i \in I \qquad \forall j \in 1..n \qquad \vdash^{w}_{\text{pat}} p_i^j : T_i^j \rhd \Gamma_i^j \qquad \Delta \cdot \left(\Gamma, \Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^{w} S_i \rhd t_i : T \lhd S'}{\Delta \cdot \Gamma \vdash^{w} \&\left\{?\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i\in I} \rhd \mathtt{receive\ do}\ \left(\{:\mathtt{l}_i, \widetilde{p_i}\} \to t_i\right)_{i\in I} \mathtt{end} : T \lhd S'}$$

The `receive` construct is typechecked using the $[\textsc{tBranch}]$ rule. It expects an (external) branching session type $\&\{\ldots\}$, where each branch in the session type must match with a corresponding branch in the `receive` construct, where *both* the labels ($\mathtt{l}_i$) and payload types ($\widetilde{T_i}$) correspond. The types within each `receive` branch are computed using the pattern typing judgement, $\vdash^{w}_{\text{pat}} p : T \rhd \Gamma$, which assigns types to variables present in patterns (see Appendix B.3). Each `receive` branch is then checked w.r.t. the common type T and a common residual session type S'.

$$[\textsc{tChoice}]\ \frac{\exists i \in I \qquad \mathtt{l} = \mathtt{l}_i \qquad \forall j \in 1..n \qquad \Gamma \vdash_{\text{exp}} e_j : T_i^j}{\Delta \cdot \Gamma \vdash^{w} \oplus\left\{!\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i\in I} \rhd \mathtt{send}\left(w, \{:\mathtt{l}, e_1, \ldots, e_n\}\right) : \left\{\mathtt{atom}, T_i^1, \ldots, T_i^n\right\} \lhd S_i}$$

The rule $[\textsc{tChoice}]$ typechecks the sending of messages. This rule requires an internal choice session type $\oplus\{\ldots\}$, where the label tagging the message to be sent must match with one of the labels ($\mathtt{l}_i$) offered by the session choice. The message payloads must also match with the corresponding types associated with the label ($\widetilde{T_i}$ of $\mathtt{l}_i$) stated via the expression typing judgement $\Gamma \vdash_{\text{exp}} e : T$ (see Appendix B.2). The typing rule also checks the *pid* of the addressee of the `send` statement which must match with the dual *pid* (w) states in the judgment itself to ensure that messages are only sent to the correct addressee.

$$[\textsc{tRecKnownCall}]\ \frac{\begin{array}{c}\Delta\,(f/n) = S \qquad \forall i \in 2..n \cdot \left\{\Gamma \vdash_{\text{exp}} e_i : T_i\right\} \\ \Sigma\,(f/n) = \Omega \qquad \Omega.\mathtt{return\_type} = T \qquad \Omega.\mathtt{param\_types} = \widetilde{T}\end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd f\,(w, e_2, \ldots, e_n) : T \lhd \mathsf{end}}$$

Since public functions are decorated with a session type explicitly using the `@session` (or `@dual`) annotation, they are listed in $\mathbf{dom}(\Delta)$. Calls to public functions are typechecked using the $[\textsc{tRecKnownCall}]$ rule, which verifies that the expected initial session type is equivalent to the function's *known* session type (S) obtained from the *session typing* environment, *i.e.*, $\Delta\,(f/n) = S$. Without typechecking the function's body, which is done in rule $[\textsc{tModule}]$, this rule ensures that the parameters have the correct types (using the expression typing rules). From the check performed in rule $[\textsc{tModule}]$, it can also safely assume that this session type S is fully consumed, thus the residual type becomes end. Rule $[\textsc{tRecKnownCall}]$ also ensures that the *pid* (w) is preserved during a function call, by requiring it to be passed as a parameter and comparing it to the expected dual *pid* (*i.e.*, $\Delta \cdot \Gamma \vdash^{\boxed{w}} S \rhd f\left(\boxed{w}, \ldots\right) : T \lhd \mathsf{end}$).

$$[\textsc{tRecUnknownCall}]\ \frac{\begin{array}{c}\Sigma(f/n)=\Omega \qquad f/n\notin \mathbf{dom}(\Delta) \qquad \Omega.\mathtt{dual}=y \\ \Omega.\mathtt{params}=\widetilde{x} \quad \Omega.\mathtt{param\_type}=\widetilde{T} \quad \Omega.\mathtt{body}=t \quad \Omega.\mathtt{return\_type}=T \\ (\Delta, f/n:S)\cdot\left(\Gamma, y:\mathsf{pid}, \widetilde{x}:\widetilde{T}\right)\vdash^{y} S\rhd t:T\lhd S' \qquad \forall i\in 2..n\cdot\left\{\Gamma\vdash_{\exp} e_i:T_i\right\}\end{array}}{\Delta\cdot\Gamma\vdash^{w} S\rhd f\,(w,\, e_2,\, \ldots,\, e_n):T\lhd S'}$$

Contrastingly, a call to a (private) function, f/n, with an *unknown* session type associated to it is type-checked using the [tRecUnknownCall] rule. As in the other rule, it ensures that parameters have the correct types ($\Gamma\vdash_{\exp} e_i : T_i$). However, it also analyses the function's body t (obtained from Σ) with respect to the session type S inherited from the initial session type of the call, Furthermore, this session type is appended to the *session typing* environment Δ for future reference, *i.e.,* $\Delta' = (\Delta, f/n : S)$ which allows it to handle recursive calls to itself; should the function be called again, rule [tRecKnownCall] is used thus bypassing the need to re-analyse its body.

2.5 Elixir System

The ElixirST provides a bespoke spawning function called `session/4` which allows the initiation of two concurrent processes executing in tandem as part of a session. This `session/4` function takes two pairs of arguments: two references of function names (that will be spawned), along with their list of arguments. Its participant creation flow is shown in Figure 3. Initially the actor (pre-server) is spawned, passing its *pid* (ι_{server}) to the second spawned actor (pre-client). Then, pre-client relays back its *pid* (ι_{client}) to pre-server. In this way, both actors participating in a session become aware of each other's *pids*. From this point onwards, the two actors execute their respective function to behave as the participants in the binary session; the first argument of each running function is initiated to the respective *pid* of the other participant. Figure 3 shows that the server process executes the body t, where it has access to the mailbox $\mathscr{M}$. As it executes, messages may be sent or received (shown by the action α) and stored in the mailbox $\mathscr{M}'$. The specific working of these transitions is explained in the following section.

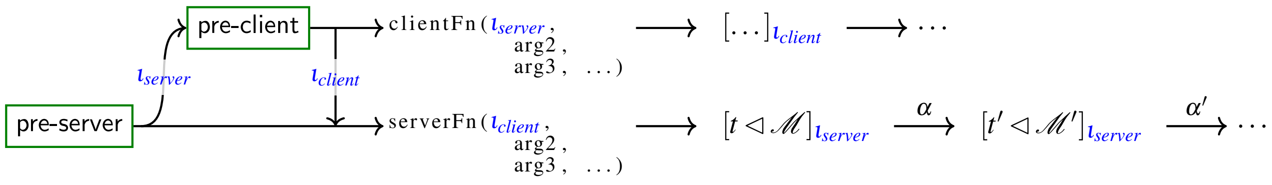

pre-client
pre-server
ι_{server}
ι_{client}
clientFn(ι_{server}, arg2, arg3, ...)
serverFn(ι_{client}, arg2, arg3, ...)
$[\ldots]_{\iota_{client}}$
$[t\lhd\mathscr{M}]_{\iota_{server}}$
$[t'\lhd\mathscr{M}']_{\iota_{server}}$
α
α'

Figure 3: Spawning two processes (green boxes represent *spawned* concurrent processes)

3 Operational Semantics

We describe the operational semantics of the Elixir language subset of Figure 2 as a *labelled transition system* (LTS) [18] describing how a handler process within a session executes while interacting with the session client, as outlined in Figure 1. The transitions $t \xrightarrow{\alpha} t'$ describes the fact that a handler process in state t performs an execution step to transition to the new state t' while producing action α as a side-effect. External actions are visible by, and bear an effect on the client, whereas internal actions do not. In our case, an action α can take the following forms:

$$\begin{array}{rcll}
\alpha \in \textsc{Act} & ::= & \iota!\{\texttt{:l},\, \widetilde{v}\} & \text{Output message to } \iota \text{ tagged as } \texttt{:l} \text{ with payload } \widetilde{v} \\
& \mid & ?\{\texttt{:l},\, \widetilde{v}\} & \text{Input message tagged as } \texttt{:l} \text{ with payload } \widetilde{v} \\
& \mid & f/n & \text{Call function } f \text{ with arity } n \\
& \mid & \tau & \text{Internal reduction step}
\end{array}
\quad
\begin{array}{l}
\left.\right\} \text{external action} \\
\left.\right\} \text{internal action}
\end{array}$$

Both output and input actions constitute external actions that affect either party in a session; the type system from Section 2.4 disciplines these external actions. Internal actions, include *silent* transition (τ) and function calls (f/n); although the latter may be formalised as a silent action, the decoration facilitates our technical development. We note that, function calls can only transition subject to a well-formed *function information* environment (Σ), which contains details about all the functions available in the module. Since Σ remains static during transitions, we leave it implicit in the transitions rules.

The transitions are defined by the *term* transition rules listed in Figure 4. Rules [RLET$_1$] and [RLET$_2$] deal with the evaluation of a *let* statement, $x = t_1;\, t_2$ modelling a *call-by-value* semantic, where the first term t_1 has to transition fully to a value before being substituted for x in t_2 denoted as $[^v/_x]$ (or $[^{v_1,v_2}/_{x_1,x_2}]$ for multiple substitutions). The *send* statement, $\texttt{send}(\iota, \{\texttt{:l}, e_1,\, \dots,\, e_n\})$, evaluates by first reducing each

$$\boxed{t \xrightarrow[\Sigma]{\alpha} t'} \qquad [\textsc{rLet}_1]\ \frac{t_1 \xrightarrow{\alpha} t_1'}{x = t_1;\, t_2 \xrightarrow{\alpha} x = t_1';\, t_2} \qquad [\textsc{rLet}_2]\ \frac{}{x = v;\, t \xrightarrow{\tau} t\,[^v/_x]}$$

$$[\textsc{rChoice}_1]\ \frac{e_k \to e_k'}{\texttt{send}(\iota, \{\texttt{:l}, v_1,\, \dots,\, v_{k-1},\, e_k,\, \dots,\, e_n\}) \xrightarrow{\tau} \texttt{send}\left(\iota, \left\{\texttt{:l}, v_1,\, \dots,\, v_{k-1},\, e_k',\, \dots,\, e_n\right\}\right)}$$

$$[\textsc{rChoice}_2]\ \frac{}{\texttt{send}(\iota, \{\texttt{:l}, v_1,\, \dots,\, v_n\}) \xrightarrow{\iota!\{\texttt{:l},\, v_1,\, \dots,\, v_n\}} \{\texttt{:l}, v_1,\, \dots,\, v_n\}}$$

$$[\textsc{rBranch}]\ \frac{\exists j \in I \qquad \texttt{l}_j = \texttt{l} \qquad \mathbf{match}(\widetilde{p}_j,\, v_1,\, \dots,\, v_n) = \sigma}{\texttt{receive do } (\{\texttt{:l}_i, \widetilde{p}_i\} \to t_i)_{i\in I}\texttt{end} \xrightarrow{?\{\texttt{:l},\, v_1,\, \dots,\, v_n\}} t_j\sigma}$$

$$[\textsc{rCall}_1]\ \frac{e_k \to e_k'}{f\,(v_1,\, \dots,\, v_{k-1},\, e_k,\, \dots,\, e_n) \xrightarrow{\tau} f\left(v_1,\, \dots,\, v_{k-1},\, e_k',\, \dots,\, e_n\right)}$$

$$[\textsc{rCall}_2]\ \frac{\Sigma\,(f/n) = \Omega \qquad \Omega.\texttt{body} = t \qquad \Omega.\texttt{params} = x_2,\, \dots,\, x_n \qquad \Omega.\texttt{dual} = y}{f\,(\iota, v_2,\, \dots,\, v_n) \xrightarrow{f/n} t\,[^\iota/_y]\,[^{v_2,\, \dots,\, v_n}/_{x_2,\, \dots,\, x_n}]}$$

$$[\textsc{rCase}_1]\ \frac{e \to e'}{\texttt{case } e \texttt{ do } (p_i \to t_i)_{i\in I}\texttt{end} \xrightarrow{\tau} \texttt{case } e' \texttt{ do } (p_i \to t_i)_{i\in I}\texttt{end}}$$

$$[\textsc{rCase}_2]\ \frac{\exists j \in I \qquad \mathbf{match}(p_j, v) = \sigma}{\texttt{case } v \texttt{ do } (p_i \to t_i)_{i\in I}\texttt{end} \xrightarrow{\tau} t_j\sigma} \qquad [\textsc{rExpression}]\ \frac{e \to e'}{e \xrightarrow{\tau} e'}$$

Figure 4: Term transition semantic rules

$$\boxed{e \to e'} \quad [\text{REOPERATION}_1]\ \frac{e_1 \to e_1'}{e_1 \diamond e_2 \to e_1' \diamond e_2} \qquad [\text{REOPERATION}_2]\ \frac{e_2 \to e_2'}{v_1 \diamond e_2 \to v_1 \diamond e_2'}$$

$$[\text{REOPERATION}_3]\ \frac{v = v_1 \diamond v_2}{v_1 \diamond v_2 \to v} \quad [\text{RENOT}_1]\ \frac{e \to e'}{\texttt{not}\ e \to e'} \quad [\text{RENOT}_2]\ \frac{v' = \neg v}{\texttt{not}\ v \to v'}$$

$$[\text{RELIST}_1]\ \frac{e_1 \to e_1'}{[e_1 \mid e_2] \to [e_1' \mid e_2]} \qquad [\text{RELIST}_2]\ \frac{e_2 \to e_2'}{[v_1 \mid e_2] \to [v_1 \mid e_2]}$$

$$[\text{RETUPLE}]\ \frac{e_k \to e_k'}{\{v_1,\ \dots,\ v_{k-1},\ e_k,\ \dots,\ e_n\} \to \{v_1,\ \dots,\ ,v_{k-1},\ e_k',\ \dots,\ e_n\}}$$

Figure 5: Expression reduction rules

part of the message to a value from left to right. This is carried out via rule $[\text{RCHOICE}_1]$ which produces no observable side-effects. When the whole message is reduced to a tuple of values $\{\texttt{:l}, v_1, \dots, v_n\}$, rule $[\text{RCHOICE}_2]$ performs the actual message sending operation. This transition produces an action $\iota!\{\texttt{:l}, v_1, \dots, v_n\}$, where the message $\{\texttt{:l}, v_1, \dots, v_n\}$ is sent to the interacting process, which has a *pid* value of ι. The operational semantics of the *receive* construct, $\texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I}\texttt{end}$, is defined by rule $[\text{RBRANCH}]$. When a message is received (*i.e.,* $\alpha = ?\{\texttt{:l},\ \widetilde{v}\}$), it is matched with a valid branch from the *receive* construct, using the label $\texttt{:l}$. Should one of the labels match ($\exists j \in I$ such that $\texttt{:l}_j = \texttt{:l}$), the payload of the message ($\widetilde{v}$) is compared to the corresponding patterns in the selected branch ($\widetilde{p_j}$) using $\mathbf{match}(\widetilde{p_j}, \widetilde{v})$. If the values match with the pattern, the **match** function (Definition 3.1) produces the substitutions σ, mapping the matched variables in the pattern $\widetilde{p_j}$ to values from $\widetilde{v}$. This substitution σ is then used to instantiate the free variables in continuation branch t_j.

Definition 3.1 *(Pattern Matching)***.** The **match** function pairs patterns with a corresponding value, resulting in a sequence of substitutions (called σ), *e.g.*, $\mathbf{match}(p, v) = [^{v_1}/_{x_1}]\,[^{v_2}/_{x_2}] = [^{v_1,v_2}/_{x_1,x_2}]$. Note that, a sequence of **match** outputs are combined together, where the empty substitutions (*i.e.,* $[\,]$) are ignored. The match function builds a meta-list of substitutions, which is a different form of lists defined by the Elixir syntax in Figure 2.

$$\mathbf{match}(\widetilde{p}, \widetilde{v}) \stackrel{\text{def}}{=} \mathbf{match}(p_1, v_1),\ \dots,\ \mathbf{match}(p_n, v_n) \qquad \textit{where } \widetilde{p} = p_1,\ \dots,\ p_n \textit{ and } \widetilde{v} = v_1,\ \dots,\ v_n$$

$$\mathbf{match}(p, v) \stackrel{\text{def}}{=} \begin{cases} [\,] & p = b, v = b \textit{ and } p = v \\ [^{v}/_{x}] & p = x \\ \mathbf{match}(w_1, v_1),\ \mathbf{match}(w_2, v_2) & p = [w_1 \mid w_2], v = [v_1 \mid v_2] \\ \mathbf{match}(w_1, v_1),\ \dots,\ \mathbf{match}(w_n, v_n) & p = \{w_1,\ \dots,\ w_n\} \textit{ and } \\ & \quad v = \{v_1,\ \dots,\ v_n\} \end{cases} \blacksquare$$

Example 3.1. For the pattern $p_1 = \{x, 2, y\}$ and the value tuple $v_1 = \{8, 2, \mathit{true}\}$, $\mathbf{match}(p_1, v_1) = \sigma$ where $\sigma = [^{8}/_{x}]\,[^{\mathit{true}}/_{y}]$ (written also as $\sigma = [^{8,\mathit{true}}/_{x,y}]$). However for pattern $p_2 = \{x, 2, \mathit{false}\}$, the operation

$\mathbf{match}(p_2, v_1)$ fails, since p_2 expects a *false* value as the third element, but finds a *true* value instead. ■

Using rule [RCALL$_1$] from Figure 4, a function call is evaluated by first reducing all of its parameters to a value, using the expression reduction rules in Figure 5; again this models a call-by-value semantics. Once all arguments have been fully reduced, [RCALL$_2$], the implicit environment Σ is queried for function f with arity n to fetch the function's parameter names and body. This results in a transition to the function body with its parameters instantiated accordingly, $t\,[\iota/y]\,[v_2,\,\ldots,\,v_n/x_2,\,\ldots,\,x_n]$, decorated by the function name, *i.e.,* $\alpha = f/n$. Along the same lines a case construct first reduces the expression which is being matched using rule [RCASE$_1$]. Then, rule [RCASE$_2$] matches the value with the correct branch, using the **match** function, akin to [RBRANCH]. Whenever a term consists solely of an expression, it silently reduces using [REXPRESSION] using the expression reduction rules $e \to e'$ of Figure 5. These are fairly standard.

4 Session Fidelity

We validate the static properties imposed by the ElixirST type system [32], overviewed in Section 2.4, by establishing a relation with the runtime behaviour of a typechecked Elixir program, using the transition semantics defined in Section 3. Broadly, we establish a form of *type preservation*, which states that if a well-typed term transitions, the resulting term then remains well-typed [27]. However, our notion of type preservation, needs to be stronger to also take into account *(i)* the side-effects produced by the execution; and *(ii)* the progression of the execution with respect to protocol expressed as a session type. Following the long-standing tradition in the session type community, these two aspects are captured by the refined preservation property called *session fidelity*. This property ensures that: *(i)* the communication action produced as a result of the execution of the typed process is one of the actions allowed by the current stage of the protocol; and that *(ii)* the resultant process following the transition is still well-typed w.r.t. the remaining part of the protocol that is still outstanding.

Before embarking on the proof for session fidelity, we prove an auxiliary proposition that acts as a sanity check for our operational semantics. We note that the operational semantics of Section 3 assumes that only *closed* programs are executed; an *open* program (*i.e.,* a program containing free variables) is seen as an incomplete program that cannot execute correctly due to missing information. To this end, Proposition 1 ensures that a closed term *remains closed* even after transitioning.

Proposition 1 (Closed Term)**.** *If* $\boldsymbol{fv}(t) = \emptyset$ *and* $t \xrightarrow{\alpha} t'$*, then* $\boldsymbol{fv}(t') = \emptyset$

Proof. By induction on the structure of t. Refer to Appendix C.1 for details. □

The statement of the session fidelity property relies on the definition of a partial function called **after** (Definition 4.1), which takes a session type and an action as arguments and returns another session type as a result. This function serves two purposes: (a) the function $\mathbf{after}(S, \alpha)$ is only defined for actions α that are (immediately) permitted by the protocol S, which allows us to verify whether a term transition step violated a protocol or not; and (b) since S describes the current stage of the protocol to be followed, we need a way to evolve this protocol to the next stage should α be a permitted action, and this is precisely S', the continuation session type returned where $\mathbf{after}(S, \alpha) = S'$.

Definition 4.1 *(After Function)***.** The **after** function is partial function defined for the following cases:

$$\mathbf{after}(S,\tau) \stackrel{\text{def}}{=} S$$
$$\mathbf{after}(S,f/n) \stackrel{\text{def}}{=} S$$
$$\mathbf{after}(\oplus\{!\mathtt{l}_i(\widetilde{T_i}).S_i\}_{i\in I}, \iota!\{\mathtt{l}_j,\widetilde{v}\}) \stackrel{\text{def}}{=} S_j \quad \text{where } j\in I$$
$$\mathbf{after}(\&\{?\mathtt{l}_i(\widetilde{T_i}).S_i\}_{i\in I}, ?\{\mathtt{l}_j,\widetilde{v}\}) \stackrel{\text{def}}{=} S_j \quad \text{where } j\in I$$

This function is undefined for all other cases. The **after** function is overloaded to range over *session typing* environments (Δ) in order to compute a new *session typing* environment given some action α and session type S:

$$\mathbf{after}(\Delta,f/n,S) \stackrel{\text{def}}{=} \Delta, f/n:S$$
$$\mathbf{after}(\Delta,\alpha,S) \stackrel{\text{def}}{=} \Delta \qquad \text{if } \alpha\neq f/n$$

Intuitively, when the action produced by the transition is f/n, the *session typing* environment is extended by the new mapping $f/n:S$. For all other actions, the *session typing* environment remains unchanged. ■

Recall that module typechecking using rule [TMODULE] entails typechecking the bodies of all the public functions w.r.t. their ascribed session type, $\Delta\cdot\left(y:\mathsf{pid},\widetilde{x}:\widetilde{T}\right)\vdash^{y}_{\Sigma} S\rhd t:T\lhd S'$ (where $S'=\mathtt{end}$ for this specific case). At runtime, a spawned client handler process in a session starts running the function body term t where the parameter variables $y,\widetilde{x}$ are instantiated with the PID of the client, say ι, and the function parameter values, say $\widetilde{v}$, respectively, $t\,[\iota/y]\,[\widetilde{v}/\widetilde{x}]$, as modelled in rule [RCALL$_2$] from Figure 4. The instantiated function body is thus closed and can be typed w.r.t. an empty variable binding environment, $\Gamma=\emptyset$. Session fidelity thus states that if a closed term t is well-typed, *i.e.,*

$$\Delta\cdot\emptyset\vdash^{w} S\rhd t:\boxed{T}\lhd S' \tag{1}$$

(where S and S' are initial and residual session types, respectively, and T is the basic expression type) and this term t transitions to a new term t' with action α, *i.e.,*

$$t\xrightarrow{\alpha} t' \tag{2}$$

the new term t' remains well-typed, *i.e.,*

$$\Delta'\cdot\emptyset\vdash^{w} S''\rhd t':\boxed{T}\lhd S' \tag{3}$$

where the evolved S'' and Δ' are computed as $\mathbf{after}(S,\alpha)=S''$ and $\mathbf{after}(\Delta,\alpha,S)=\Delta'$. This ensures that the base type of the term is preserved (note the constant type $\boxed{T}$ in eqs. (1) and (3)). Furthermore, it ascertains that the term t follows an interaction protocol starting from the initial session type S up to the residual session type S' (eq. (1)), since the updated session type S'' is defined for $\mathbf{after}(S,\alpha)$.

Theorem 2 (Session Fidelity)**.** *If $\Delta\cdot\emptyset\vdash^{w}_{\Sigma} S\rhd t:T\lhd S'$ and $t\xrightarrow[\Sigma]{\alpha} t'$, then there exists some S'' and Δ', such that $\Delta'\cdot\emptyset\vdash^{w}_{\Sigma} S''\rhd t':T\lhd S'$ for $\mathbf{after}(S,\alpha)=S''$ and $\mathbf{after}(\Delta,\alpha,S)=\Delta'$*

Proof. By induction on the typing derivation $\Delta\cdot\emptyset\vdash^{w}_{\Sigma} S\rhd t:T\lhd S'$. Refer to Appendix C.2. □

$$\begin{tikzcd} t_1 \arrow[r, "\alpha_1"] \arrow[d, dotted, no head] & t_2 \arrow[r, "\alpha_2"] \arrow[d, dotted, no head] & \cdots \arrow[r, "\alpha_n"] & v \arrow[r, "\setminus" marking] \arrow[d, dotted, no head] & {} \\ S_1 & \textbf{after}(S_1,\alpha_1) = S_2 & & \textbf{after}(S_n,\alpha_n) = \mathsf{end} & \end{tikzcd}$$

Figure 6: Repeated applications of *session fidelity*

As shown in Figure 6, by repeatedly applying Theorem 2, we can therefore conclude that all the (external) actions generated as a result of a computation (*i.e.,* sequence of transition steps) must all be actions that follow the protocol described by the initial session type. Since public functions are always typed with a residual session type end, certain executions could reach the case where the outstanding session is updated to end as well, *i.e.,* $\textbf{after}(S_n, \alpha_n) = \mathsf{end}$. In such a case, we are guaranteed that the term will not produce further side-effects, as in the case of Figure 6 where the term is reduced all the way down to some value, v.

Example 4.1. We consider a concrete example to show the importance of session fidelity. The function called `pinger/1` is able to send `ping` and receive `pong` repeatedly.

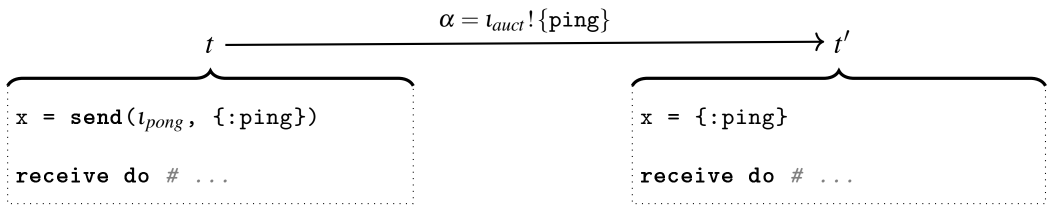
```
@session "X = !ping().?pong().X"
def pinger(pid) do
  x = send(pid, {:ping})

  receive do
    {:pong} -> IO.puts("Received pong.")
  end
  pinger(pid)
end
```

This function adheres to the following protocol:

$$\mathsf{X} = !\mathtt{ping}().?\mathtt{pong}().\mathsf{X}$$

A process evaluating the function `pinger` executes by first sending a message containing a `ping` label to the interacting processes' *pid* (ι_{pong}), as shown below.



```
x = send(ιpong, {:ping})

receive do # ...
```

```
x = {:ping}

receive do # ...
```

As the process evaluates, the initial term t transitions to t', where it sends a message as a side-effect. This side-effect is denoted as an action α, where $\alpha = \iota_{pong}!\{\mathtt{ping}\}$. By the After Function Definition, X evolves to a new session type X':

$$\mathsf{X'} = \textbf{after}(!\mathtt{ping}().?\mathtt{pong}().\mathsf{X}, \alpha) = ?\mathtt{pong}().\mathsf{X}$$

For t' to remain well-typed, it must now match with the evolved session type X', where it has to be able to receive a message labelled `pong`, before recursing. Although the process keeps executing indefinitely, by the session fidelity property, we know that each step of execution will be in line with the original protocol. ■

5 Related Work

In this section, we compare ElixirST with other type systems and implementations.

Type Systems for Elixir Cassola *et al.* [4, 5] presented a gradual type system for Elixir. It statically typechecks the functional part of Elixir modules, using a gradual approach, where some terms may be left with an unknown expression type. In contrast to ElixirST, Cassola *et al.* analyse directly the unexpanded Elixir code which results in more explicit typechecking rules. Also, they focus on the static type system without formulating the operational semantics.

Another static type-checker for Elixir is *Gradient* [8]. It is a wrapper for its Erlang counterpart tool and takes a similar approach to [5], where gradual types are used. Another project, *TypeCheck* [35], adds dynamic type validations to Elixir programs. *TypeCheck* performs runtime typechecking by wrapping checks around existing functions. *Gradient* and *TypeCheck* are provided as an implementation only, without any formal analysis. In contrast to ElixirST, the discussed type-checkers [5, 8, 35] analyse the sequential part of the Elixir language omitting any checks related to message-passing between processes.

Some implementations aim to check issues related to message-passing. Harrison [11] statically checks Core Erlang for such issues. For instance, it detects orphan messages (*i.e.,* messages that will never be received) and unreachable receive branches. Harrison [12] extends [11] to add analyse Erlang/OTP behaviours (*e.g.*, `gen_server`, which structures processes in a hierarchical manner) by injecting runtime checks in the code. Compared to our work, [11, 12] perform automatic analysis of the implementation, however they do not verify communication with respect to a general protocol (*e.g.*, session types).

Another type system for Erlang was presented Svensson *et al.* [31]. Their body of work covers a larger subset of Erlang to what would be its equivalent in Elixir covered by our work. Moreover, its multi-tiered semantics captures an LTS defined over systems of concurrent actors. Although we opted for a smaller subset, we go beyond the pattern matching described by Svensson *et al.* since we perform a degree of typechecking for base types (*e.g.* in the premise of [tBranch]).

Session Type Systems. Closest to our work is [23], where Mostrous and Vasconcelos introduced session types to a fragment of Core Erlang, a dynamically typed language linked to Elixir. Their type system tags each message exchanged with a unique reference. This allows multiple sessions to coexist, since different messages could be matched to the corresponding session, using correlation sets. Mostrous and Vasconcelos takes a more theoretic approach, so there is no implementation for [23]. Their type system guarantees *session fidelity* by inspecting the processes' mailboxes where, at termination, no messages should be left unprocessed in their mailboxes. Our work takes a more limited but pragmatic approach, where we introduce session types for functions within a module. Furthermore, we offer additional features, including variable binding (*e.g.*, in let statements), expressions (*e.g.*, addition operation), inductive types (*e.g.*, tuples and lists), infinite computation via recursion and explicit protocol definition.

A session-based runtime monitoring tool for python was initially presented by Neykova and Yoshida [24, 25]. They use the Scribble [15] language to write *multiparty* session type (MPST) [16] protocols, which are then used to monitor the processes' actions. Different processes are ascribed a role (defined in the MPST protocol) using function decorators (akin to our function annotations). Similar to [24, 25], Fowler [9] presented an MPST implementation for Erlang. This implementation uses Erlang/OTP behaviours (*e.g.*, `gen_server`), which take into account Erlang's *let it crash* philosophy, where processes may fail while executing. In contrast, although our work accepts a more limited language, ElixirST provides static guarantees where issues are flagged at pre-deployment stages, rather than flagging them at runtime.

Scalas and Yoshida [29] applied binary session types to the Scala language, where session types are abstracted as Scala classes. Session fidelity is ensured using Scala's compiler, which complains if an implementation does not follow its ascribed protocol. Linearity checks are performed at runtime, which ensure that an implementation fully exhausts its protocol exactly once. Bartolo Burlò *et al.* [3] extended the aforementioned work [29], to monitor one side of an interaction statically and the other side dynamically using runtime monitors.

Harvey *et al.* [13] presented a new actor-based language, called EnsembleS, which offers session types as a native feature of the language. EnsembleS statically verifies implementations with respect to session types, while still allowing for adaptation of *new* actors at runtime, given that the actors obey a known protocol. Thus, actors can be terminated and discovered at runtime, while still maintaining static correctness.

There have been several binary [17, 19] and multiparty [6, 20] session type implementations for Rust. These implementations exploit Rust's affine type system to guarantee that channels mirror the actions prescribed by a session type. Padovani [26] created a binary session type library for OCaml to provide static communication guarantees. This project was extended [22] to include dynamic contract monitoring which flags violations at runtime. The approaches used in the Rust and OCaml implementations rely heavily on type-level features of the language, which do not readily translate to the dynamically typed Elixir language.

6 Conclusion

In this work we established a correspondence between the ElixirST type system [32] and the runtime behaviour of a client handler running an Elixir module function that has been typechecked w.r.t. its session type protocol. In particular, we showed that this session-based type system observes the standard *session fidelity* property, meaning that processes executing a typed function *always* follow their ascribed protocols at runtime. This property provides the necessary underlying guarantees to attain various forms of communication safety, whereby should two processes following mutually compatible protocols (*e.g.* S and its dual $\bar{S}$), they avoid certain communication errors (*e.g.*, a send statement without a corresponding receive construct). An extended version of this work can be found in the technical report [33].

Future work. There are a number of avenues we intend to pursue. One line of investigation is the augmentation of protocols that talk about multiple entry points to a module perhaps from the point of view of a client that is engaged in multiple sessions at one time, possibly involving multiple modules. The obvious starting points to look at here are the well-established notions of multiparty session types [16, 30] or the body of work on intuitionistic session types organising processes hierarchically [2, 28]. Another natural extension to our work would be to augment our session type protocol in such a way to account for process failure and supervisors, which is a core part of the Elixir programming model. For this, we will look at previous work on session type extensions that account for failure [13]. Finally, we also plan to augment our session typed protocols to account for resource usage and cost, along the lines of [7, 10].

Acknowledgements. We thank Simon Fowler and Matthew Alan Le Brun for their encouragement and suggestions for improvement on earlier versions of the work. This work has been supported by the MoVeMnt project (No: 217987) of the Icelandic Research Fund and the BehAPI project funded by the EU H2020 RISE of the Marie Skłodowska-Curie action (No: 778233).

References

[1] Gul A. Agha (1990): *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, MIT Press.

[2] Stephanie Balzer & Frank Pfenning (2017): *Manifest sharing with session types*. *Proc. ACM Program. Lang.* 1(ICFP), pp. 37:1–37:29, doi:10.1145/3110281.

[3] Christian Bartolo Burlò, Adrian Francalanza & Alceste Scalas (2021): *On the Monitorability of Session Types, in Theory and Practice*. In Anders Møller & Manu Sridharan, editors: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, *LIPIcs* 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 20:1–20:30, doi:10.4230/LIPIcs.ECOOP.2021.20.

[4] Mauricio Cassola, Agustín Talagorria, Alberto Pardo & Marcos Viera (2020): *A Gradual Type System for Elixir*. *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*, doi:10.1145/3427081.3427084.

[5] Mauricio Cassola, Agustín Talagorria, Alberto Pardo & Marcos Viera (2022): *A gradual type system for Elixir*. *Journal of Computer Languages* 68, p. 101077, doi:10.1016/j.cola.2021.101077.

[6] Zak Cutner & Nobuko Yoshida (2021): *Safe Session-Based Asynchronous Coordination in Rust*. In Ferruccio Damiani & Ornela Dardha, editors: *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, *Lecture Notes in Computer Science* 12717, Springer, pp. 80–89, doi:10.1007/978-3-030-78142-2_5.

[7] Ankush Das, Jan Hoffmann & Frank Pfenning (2018): *Work Analysis with Resource-Aware Session Types*. In Anuj Dawar & Erich Grädel, editors: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, ACM, pp. 305–314, doi:10.1145/3209108.3209146.

[8] Erlang Solutions: *Gradient*. Available at `https://github.com/esl/gradient`.

[9] Simon Fowler (2016): *An Erlang Implementation of Multiparty Session Actors*. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight & Hugo Torres Vieira, editors: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, *EPTCS* 223, pp. 36–50, doi:10.4204/EPTCS.223.3.

[10] Adrian Francalanza, Edsko de Vries & Matthew Hennessy (2014): *Compositional Reasoning for Explicit Resource Management in Channel-Based Concurrency*. *Log. Methods Comput. Sci.* 10(2), doi:10.2168/LMCS-10(2:15)2014.

[11] Joseph Harrison (2018): *Automatic detection of Core Erlang message passing errors*. In Natalia Chechina & Adrian Francalanza, editors: *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, ICFP 2018, St. Louis, MO, USA, September 23-29, 2018*, ACM, pp. 37–48, doi:10.1145/3239332.3242765.

[12] Joseph Harrison (2019): *Runtime Type Safety for Erlang/OTP Behaviours*. In Adrian Francalanza & Viktória Fördós, editors: *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2019, Berlin, Germany, August 18, 2019*, ACM, pp. 36–47, doi:10.1145/3331542.3342571.

[13] Paul Harvey, Simon Fowler, Ornela Dardha & Simon J. Gay (2021): *Multiparty Session Types for Safe Runtime Adaptation in an Actor Language*. In Anders Møller & Manu Sridharan, editors: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, *LIPIcs* 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:30, doi:10.4230/LIPIcs.ECOOP.2021.10.

[14] Carl Hewitt, Peter Boehler Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *IJCAI*, William Kaufmann, pp. 235–245.

[15] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen & Nobuko Yoshida (2011): *Scribbling Interactions with a Formal Foundation*. In Raja Natarajan & Adegboyega K. Ojo, editors: *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, *Lecture Notes in Computer Science* 6536, Springer, pp. 55–75, doi:10.1007/978-3-642-19056-8_4.

[16] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[17] Thomas Bracht Laumann Jespersen, Philip Munksgaard & Ken Friis Larsen (2015): *Session types for Rust*. In Patrick Bahr & Sebastian Erdweg, editors: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, ACM, pp. 13–22, doi:10.1145/2808098.2808100.

[18] Robert M. Keller (1976): *Formal Verification of Parallel Programs*. *Commun. ACM* 19(7), pp. 371–384, doi:10.1145/360248.360251.

[19] Wen Kokke (2019): *Rusty Variation: Deadlock-free Sessions with Failure in Rust*. In Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou & Alceste Scalas, editors: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, *EPTCS* 304, pp. 48–60, doi:10.4204/EPTCS.304.4.

[20] Nicolas Lagaillardie, Rumyana Neykova & Nobuko Yoshida (2020): *Implementing Multiparty Session Types in Rust*. In Simon Bliudze & Laura Bocchi, editors: *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, *Lecture Notes in Computer Science* 12134, Springer, pp. 127–136, doi:10.1007/978-3-030-50029-0_8.

[21] Tobias Lindahl & Konstantinos Sagonas (2006): *Practical type inference based on success typings*. In Annalisa Bossi & Michael J. Maher, editors: *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, ACM, pp. 167–178, doi:10.1145/1140335.1140356.

[22] Hernán C. Melgratti & Luca Padovani (2017): *Chaperone Contracts for Higher-Order Sessions*. *Proc. ACM Program. Lang.* 1(ICFP), pp. 35:1–35:29, doi:10.1145/3110279.

[23] Dimitris Mostrous & Vasco Thudichum Vasconcelos (2011): *Session Typing for a Featherweight Erlang*. In Wolfgang De Meuter & Gruia-Catalin Roman, editors: *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland,*

June 6-9, 2011. Proceedings, *Lecture Notes in Computer Science* 6721, Springer, pp. 95–109, doi:10.1007/978-3-642-21464-6_7.

[24] Rumyana Neykova & Nobuko Yoshida (2014): *Multiparty Session Actors*. In Alastair F. Donaldson & Vasco T. Vasconcelos, editors: *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014*, *EPTCS* 155, pp. 32–37, doi:10.4204/EPTCS.155.5.

[25] Rumyana Neykova & Nobuko Yoshida (2017): *Multiparty Session Actors*. *Log. Methods Comput. Sci.* 13(1), doi:10.23638/LMCS-13(1:17)2017.

[26] Luca Padovani (2017): *A Simple Library Implementation of Binary Sessions*. *J. Funct. Program.* 27, p. e4, doi:10.1017/S0956796816000289.

[27] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[28] Klaas Pruiksma & Frank Pfenning (2022): *Back to futures*. *Journal of Functional Programming* 32, p. e6, doi:10.1017/S0956796822000016.

[29] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In Shriram Krishnamurthi & Benjamin S. Lerner, editors: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, *LIPIcs* 56, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[30] Alceste Scalas, Nobuko Yoshida & Elias Benussi (2019): *Verifying message-passing programs with dependent behavioural types*. In Kathryn S. McKinley & Kathleen Fisher, editors: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, ACM, pp. 502–516, doi:10.1145/3314221.3322484.

[31] Hans Svensson, Lars-Åke Fredlund & Clara Benac Earle (2010): *A unified semantics for future Erlang*. In Scott Lystig Fritchie & Konstantinos Sagonas, editors: *Proceedings of the 9th ACM SIGPLAN workshop on Erlang, Baltimore, Maryland, USA, September 30, 2010*, ACM, pp. 23–32, doi:10.1145/1863509.1863514.

[32] Gerard Tabone & Adrian Francalanza (2021): *Session Types in Elixir*. In Elias Castegren, Joeri De Koster & Simon Fowler, editors: *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2021, Virtual Event / Chicago, IL, USA, 17 October 2021*, ACM, pp. 12–23, doi:10.1145/3486601.3486708.

[33] Gerard Tabone & Adrian Francalanza (2022): *Static Checking of Concurrent Programs in Elixir Using Session Types*. Technical Report, University of Malta, Msida, Malta. Available at `https://gtabone.page.link/V9Hh`.

[34] Dave Thomas (2018): *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. Pragmatic Bookshelf.

[35] Wiebe-Marten Wijnja: *TypeCheck: Fast and flexible runtime type-checking for your Elixir projects*. Available at `https://github.com/Qqwy/elixir-type_check`.

Appendix

A Additional Definitions

In this appendix, we formalise some auxiliary definitions that were used in Sections 2–4.

Definition A.1 *(Free Variables)*. The set of free variables is defined inductively as:

$$\mathbf{fv}(e) \stackrel{\text{def}}{=} \begin{cases} \{x\} & e = x \\ \emptyset & e = b \\ \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) & e = e_1 \diamond e_2 \text{ or } e = [e_1 \mid e_2] \\ \mathbf{fv}(e') & e = \texttt{not}\ e' \\ \cup_{i \in 1..n} \mathbf{fv}(e_i) & e = \{e_1, \ldots, e_n\} \end{cases}$$

$$\mathbf{fv}(t) \stackrel{\text{def}}{=} \begin{cases} \mathbf{fv}(t_1) \cup (\mathbf{fv}(t_2) \setminus \{x\}) & t = (x = t_1;\, t_2) \\ \cup_{i \in 1..n} \mathbf{fv}(e_i) \cup \mathbf{fv}(w) & t = \texttt{send}\,(w, \{\texttt{:l}, e_1, \ldots, e_n\}) \\ \cup_{i \in I} [\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i})] & t = \texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I} \texttt{end} \\ \cup_{i \in 2..n} \mathbf{fv}(e_i) \cup \mathbf{fv}(w) & t = f\,(w, e_2, \ldots, e_n) \\ \cup_{i \in I} [\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i)] \cup \mathbf{fv}(e) & t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I} \texttt{end} \end{cases} \qquad \blacksquare$$

Definition A.2 *(Bound Variables)*.

$$\mathbf{bv}(t) \stackrel{\text{def}}{=} \begin{cases} \emptyset & t = e \text{ or } t = \texttt{send}\,(w, \{\texttt{:l}, \widetilde{e}\}) \text{ or } t = f\,(\widetilde{e}) \\ \{x\} \cup \mathbf{bv}(t_1) \cup \mathbf{bv}(t_2) & t = (x = t_1;\, t_2) \\ \cup_{i \in I} [\mathbf{bv}(t_i) \cup \mathbf{vars}(\widetilde{p_i})] & t = \texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I} \texttt{end} \\ \cup_{i \in I} [\mathbf{bv}(t_i) \cup \mathbf{vars}(p_i)] & t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I} \texttt{end} \end{cases} \qquad \blacksquare$$

Definition A.3 *(Variable Substitution)*.

$$e\,[{}^{v}\!/\!_{x}] \stackrel{\text{def}}{=} \begin{cases} v & e = x \\ y & e = y,\ y \neq x \\ b & e = b \\ e_1\,[{}^{v}\!/\!_{x}] \diamond e_2\,[{}^{v}\!/\!_{x}] & e = e_1 \diamond e_2 \\ \texttt{not}\ (e'\,[{}^{v}\!/\!_{x}]) & e = \texttt{not}\ e' \\ [e_1\,[{}^{v}\!/\!_{x}] \mid e_2\,[{}^{v}\!/\!_{x}]] & e = [e_1 \mid e_2] \\ \{e_1\,[{}^{v}\!/\!_{x}], \ldots, e_n\,[{}^{v}\!/\!_{x}]\} & e = \{e_1, \ldots, e_n\} \end{cases}$$

$$t\,[{}^{v}\!/\!_{x}] \stackrel{\text{def}}{=} \begin{cases} \texttt{send}\,(w\,[{}^{v}\!/\!_{x}], \{\texttt{: l}, e_1\,[{}^{v}\!/\!_{x}], \ldots, e_n\,[{}^{v}\!/\!_{x}]\}) & t = \texttt{send}\,(w, \{\texttt{: l}, e_1, \ldots, e_n\}) \\ \texttt{receive do}\ (\{\texttt{l}_i, \widetilde{p_i}\} \to t_i\,[{}^{v}\!/\!_{x}])_{i \in I} \texttt{end} & t = \texttt{receive do}\ (\{\texttt{l}_i, \widetilde{p_i}\} \to t_i)_{i \in I} \texttt{end} \\ f\,(e_1\,[{}^{v}\!/\!_{x}], \ldots, e_n\,[{}^{v}\!/\!_{x}]) & t = f\,(e_1, \ldots, e_n) \\ \texttt{case}\ e\,[{}^{v}\!/\!_{x}]\ \texttt{do}\ (p_i \to t_i\,[{}^{v}\!/\!_{x}])_{i \in I} \texttt{end} & t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I} \texttt{end} \\ y = t_1\,[{}^{v}\!/\!_{x}];\ t_2\,[{}^{v}\!/\!_{x}] & t = (y = t_1;\, t_2),\ x \neq y,\ y \neq v \end{cases} \qquad \blacksquare$$

Definition A.4 *(Type)*.

$$\textbf{type}(\mathit{boolean}) \stackrel{\text{def}}{=} \mathsf{boolean} \qquad \textbf{type}(\mathit{number}) \stackrel{\text{def}}{=} \mathsf{number}$$
$$\textbf{type}(\mathit{atom}) \stackrel{\text{def}}{=} \mathsf{atom} \qquad \textbf{type}(\iota) \stackrel{\text{def}}{=} \mathsf{pid}, \text{ where } \iota \text{ is a } \mathit{pid} \text{ instance} \qquad \blacksquare$$

Definition A.5 *(Variable Patterns)*. Computes an ordered set of variables from a given pattern p.

$$\textbf{vars}(\widetilde{p}) \stackrel{\text{def}}{=} \textbf{vars}(p_1, \ldots, p_n) \stackrel{\text{def}}{=} \textbf{vars}(p_1) \cup \cdots \cup \textbf{vars}(p_n)$$
$$\textbf{vars}(p) \stackrel{\text{def}}{=} \begin{cases} \emptyset & p = b \\ \{x\} & p = x \\ \textbf{vars}(w_1) \cup \textbf{vars}(w_2) & p = [w_1 \mid w_2] \\ \cup_{i \in 1..n} \textbf{vars}(w_i) & p = \{w_1, \ldots, w_n\} \end{cases} \qquad \blacksquare$$

Definition A.6 *(Function Details)*. We can extract function details (*i.e.,* `params`, `body`, `param_types`, `return_type`, `dual`) from a list of functions ($\widetilde{Q}$) and build a mapping, using set-comprehension, as follows. The list of functions ($\widetilde{Q}$) may consist of public (D) and private (P) functions.

$$\textbf{details}(\widetilde{Q}) \stackrel{\text{def}}{=} \left\{ f/n : \begin{bmatrix} \texttt{dual} = y,\ \texttt{params} = \widetilde{x}, \\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{return\_type} = T,\ \texttt{body} = t \end{bmatrix} \;\middle|\; \begin{bmatrix} [\texttt{@session}\ \text{“}S\text{”}] \\ \texttt{@spec}\ f\left(\mathsf{pid}, \widetilde{T}\right)\ ::\ T \\ \texttt{def}[\mathrm{p}]\ f(y, \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{Q} \right\}$$

$\blacksquare$

Definition A.7 *(Functions Names and Arity)*. This definition (**functions**()) takes the set of all public function ($\widetilde{D}$) as input, and returns a set of all public function names and their arity.

$$\textbf{functions}(\widetilde{D}) \stackrel{\text{def}}{=} \left\{ f/n \;\middle|\; \begin{bmatrix} \texttt{@session}\ldots;\ \texttt{@spec}\ldots \\ \texttt{def}\ f(y, x_2, \ldots, x_n)\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{D} \right\}$$

$\blacksquare$

Definition A.8 *(All Session Types)*. The function $\textbf{sessions}(\widetilde{D})$, returns the session type corresponding to each annotated public function.

$$\textbf{sessions}(\widetilde{D}) \stackrel{\text{def}}{=} \left\{ f/n : S \;\middle|\; \begin{bmatrix} \texttt{@session}\ \text{“}S\text{”};\ \texttt{@spec}\ldots \\ \texttt{def}\ f(y, x_2, \ldots, x_n)\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{D} \right\}$$

In case the `@dual` annotation is used instead of `@session`, the dual session type is computed automatically. $\blacksquare$

B Type System Rules

In this appendix, we present the full typing rules of the type system, adapted from [32], which were omitted from the Preliminaries Section.

B.1 Term Typing

In Section 2.4, we explained a few term typing rules, including [tBranch] and [tChoice]. In Figure 7, we present the full list of term typing rules.

$$\boxed{\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'}$$

$$[\textsc{tExpression}]\ \dfrac{\Gamma \vdash_{\text{exp}} e : T}{\Delta \cdot \Gamma \vdash^{w} S \rhd e : T \lhd S}$$

$$[\textsc{tLet}]\ \dfrac{\Delta \cdot \Gamma \vdash^{w} S \rhd t_1 : T' \lhd S'' \qquad \Delta \cdot (\Gamma, x : T') \vdash^{w} S'' \rhd t_2 : T \lhd S' \qquad x \neq w}{\Delta \cdot \Gamma \vdash^{w} S \rhd x = t_1;\, t_2 : T \lhd S'}$$

$$[\textsc{tBranch}]\ \dfrac{\forall i \in I \cdot \begin{cases} \forall j \in 1..n \cdot \left\{ \vdash^{w}_{\text{pat}} p_i^j : T_i^j \ \rhd\ \Gamma_i^j \right\} \\ \Delta \cdot \left(\Gamma, \Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^{w} S_i \rhd t_i : T \lhd S' \end{cases}}{\Delta \cdot \Gamma \vdash^{w} \&\left\{ ?\mathtt{l}_i\left(\widetilde{T_i}\right).S_i \right\}_{i \in I} \rhd \mathtt{receive\ do}\ (\{ :\mathtt{l}_i, \widetilde{p_i} \} \to t_i)_{i \in I} \mathtt{end} : T \lhd S'}$$

$$[\textsc{tChoice}]\ \dfrac{\exists i \in I \qquad \mathtt{l} = \mathtt{l}_i \qquad \forall j \in 1..n \cdot \left\{ \Gamma \vdash_{\text{exp}} e_j : T_i^j \right\}}{\Delta \cdot \Gamma \vdash^{w} \oplus\left\{ !\mathtt{l}_i\left(\widetilde{T_i}\right).S_i \right\}_{i \in I} \rhd \mathtt{send}\left(w, \{ :\mathtt{l}, e_1, \ldots, e_n \}\right) : \left\{ \mathtt{atom}, T_i^1, \ldots, T_i^n \right\} \lhd S_i}$$

$$[\textsc{tRecKnownCall}]\ \dfrac{\begin{array}{c} \Delta\,(f/n) = S \qquad \forall i \in 2..n \cdot \{ \Gamma \vdash_{\text{exp}} e_i : T_i \} \\ \Sigma\,(f/n) = \Omega \qquad \Omega.\mathtt{return\_type} = T \qquad \Omega.\mathtt{param\_types} = \widetilde{T} \end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd f\,(w, e_2, \ldots, e_n) : T \lhd \mathtt{end}}$$

$$[\textsc{tRecUnknownCall}]\ \dfrac{\begin{array}{c} \Sigma\,(f/n) = \Omega \qquad f/n \notin \mathbf{dom}(\Delta) \qquad \Omega.\mathtt{dual} = y \\ \Omega.\mathtt{params} = \widetilde{x} \quad \Omega.\mathtt{param\_type} = \widetilde{T} \quad \Omega.\mathtt{body} = t \quad \Omega.\mathtt{return\_type} = T \\ (\Delta, f/n : S) \cdot \left(\Gamma, y : \mathtt{pid}, \widetilde{x} : \widetilde{T}\right) \vdash^{y} S \rhd t : T \lhd S' \qquad \forall i \in 2..n \cdot \{ \Gamma \vdash_{\text{exp}} e_i : T_i \} \end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd f\,(w, e_2, \ldots, e_n) : T \lhd S'}$$

$$[\textsc{tCase}]\ \dfrac{\begin{array}{c} \Gamma \vdash_{\text{exp}} e : U \\ \forall i \in I \qquad \vdash^{w}_{\text{pat}} p_i : U \ \rhd\ \Gamma_i' \qquad \Delta \cdot (\Gamma, \Gamma_i') \vdash^{w} S \rhd t_i : T \lhd S' \end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd \mathtt{case}\ e\ \mathtt{do}\ (p_i \to t_i)_{i \in I} \mathtt{end} : T \lhd S'}$$

Figure 7: Term typing rules

B.2 Expression Typing

Expression are typechecked using the $\Gamma \vdash_{\text{exp}} e : T$ judgement, which states that "an expression e has type T, subject to the *variable binding* environment Γ." The expression typing rules are listed in Figure 8.

$$\boxed{\Gamma \vdash_{\text{exp}} e : T}$$

$$[\text{tTuple}]\ \frac{\forall i \in 1..n \qquad \Gamma \vdash_{\text{exp}} e_i : T_i}{\Gamma \vdash_{\text{exp}} \{e_1,\ \ldots,\ e_n\} : \{T_1,,\ \ldots,\ T_n\}}$$

$$[\text{tLiteral}]\ \frac{\mathbf{type}(b)\ =\ T \qquad b \neq []}{\Gamma \vdash_{\text{exp}} b : T} \qquad [\text{tVariable}]\ \frac{\Gamma(x) = T}{\Gamma \vdash_{\text{exp}} x : T}$$

$$[\text{tList}]\ \frac{\Gamma \vdash_{\text{exp}} e_1 : T \qquad \Gamma \vdash_{\text{exp}} e_2 : [T]}{\Gamma \vdash_{\text{exp}} [e_1 \mid e_2] : [T]} \qquad [\text{tEList}]\ \frac{}{\Gamma \vdash_{\text{exp}} [] : [T]}$$

$$[\text{tArithmetic}]\ \frac{\Gamma \vdash_{\text{exp}} e_1 : \mathsf{number} \qquad \Gamma \vdash_{\text{exp}} e_2 : \mathsf{number} \qquad \diamond \in \{+,\ -,\ *,\ /\}}{\Gamma \vdash_{\text{exp}} e_1 \diamond e_2 : \mathsf{number}}$$

$$[\text{tBoolean}]\ \frac{\Gamma \vdash_{\text{exp}} e_1 : \mathsf{boolean} \qquad \Gamma \vdash_{\text{exp}} e_2 : \mathsf{boolean} \qquad \diamond \in \{\texttt{and},\ \texttt{or}\}}{\Gamma \vdash_{\text{exp}} e_1 \diamond e_2 : \mathsf{boolean}}$$

$$[\text{tComparisons}]\ \frac{\begin{array}{c}\diamond \in \{<,\ >,\ <=,\ >=,\ ==,\ !=\}\\ \Gamma \vdash_{\text{exp}} e_1 : T \qquad \Gamma \vdash_{\text{exp}} e_2 : T\end{array}}{\Gamma \vdash_{\text{exp}} e_1 \diamond e_2 : \mathsf{boolean}} \qquad [\text{tNot}]\ \frac{\Gamma \vdash_{\text{exp}} e : \mathsf{boolean}}{\Gamma \vdash_{\text{exp}} \texttt{not}\ e : \mathsf{boolean}}$$

Figure 8: Expression typing rules

B.3 Pattern Typing

New variables may be created using patterns in the [tBranch] and [tCase] rules. These variables are matched to a type using the judgement, $\vdash^{w}_{\text{pat}} p : T \rhd \Gamma$. This judgement states that "a pattern p is matched to type T, where it produces new variables and their types are collected Γ; under the assumption that the variable containing the dual *pid*, w, remains unchanged." The pattern typing rules are found in Figure 9.

$$\boxed{\vdash^{w}_{\text{pat}} p : T \rhd \Gamma}$$

$$[\text{tpLiteral}]\ \frac{\emptyset \vdash_{\text{exp}} b : T \qquad b \neq []}{\vdash^{w}_{\text{pat}} b : T \rhd \emptyset} \qquad [\text{tpVariable}]\ \frac{x \neq w}{\vdash^{w}_{\text{pat}} x : T \rhd x : T}$$

$$[\text{tpTuple}]\ \frac{\forall i \in 1..n \qquad \vdash^{w}_{\text{pat}} w_i : T_i \rhd \Gamma_i}{\vdash^{w}_{\text{pat}} \{w_1,\ \ldots,\ w_n\} : \{T_1,\ \ldots,\ T_n\} \rhd \Gamma_1,\ \ldots,\ \Gamma_n}$$

$$[\text{tpList}]\ \frac{\vdash^{w}_{\text{pat}} w_1 : T \rhd \Gamma_1 \qquad \vdash^{w}_{\text{pat}} w_2 : [T] \rhd \Gamma_2}{\vdash^{w}_{\text{pat}} [w_1 \mid w_2] : [T] \rhd \Gamma_1, \Gamma_2} \qquad [\text{tpEList}]\ \frac{}{\vdash^{w}_{\text{pat}} [] : [T] \rhd \emptyset}$$

Figure 9: Pattern typing rules

C Proofs

In this appendix, we present the proofs of Proposition 1 and Theorem 2, which were omitted from the main text.

C.1 Proofs for Proposition 1

Before proving Proposition 1, we must analyse some properties related to closed terms, where we see how they affect variable substitutions (Definition A.3). Lemma 3 states that if a variable x does not exist inside a term t, then, if we initiate x with some value, term t must remain unaffected, *i.e.*, $t\,[{}^{v}/_{x}] = t$. Restricting this statement, Corollary 4 states that, if x is not a free variable in t, then the same result should hold. Lemma 5 consists of two statements that compare the free variables in terms (or expressions) with those that include a substitution.

Lemma 3. $x \notin \boldsymbol{fv}(t) \cup \boldsymbol{bv}(t)$ *implies* $t\,[{}^{v}/_{x}] = t$

Proof. By induction on the structure of t. □

Corollary 4. $x \notin \boldsymbol{fv}(t)$ *implies* $t\,[{}^{v}/_{x}] = t$

Proof. A consequence of Lemma 3. □

Lemma 5.

i. $x \in \boldsymbol{fv}(t)$ *implies* $\boldsymbol{fv}(t\,[{}^{v}/_{x}]) = \boldsymbol{fv}(t) \setminus \{x\}$

ii. $x \in \boldsymbol{fv}(e)$ *implies* $\boldsymbol{fv}(e\,[{}^{v}/_{x}]) = \boldsymbol{fv}(e) \setminus \{x\}$

Proof. By induction on the structures of t and e for Items *i* and *ii* respectively. □

Lemma 6. $\boldsymbol{match}(p,v) = [{}^{v_1,\,\dots,\,v_n}/_{x_1,\,\dots,\,x_n}]$, *implies* $\boldsymbol{vars}(p) = \{x_1,\,\dots,\,x_n\}$

Proof. By induction on the structure of p.

$[\boldsymbol{p = b}]$ The function $\mathbf{match}(b,v)$ succeeds only when $v = b$. So, by the **match** definition, when $v = b$,

$$\mathbf{match}(b,b) = [\,] \tag{4a}$$

By the **vars** definition, $\mathbf{vars}(b) = \emptyset$, which matches the result from eq. (4a) since no variables where substituted.

$[\boldsymbol{p = x}]$ By the **match** definition, for any v,

$$\mathbf{match}(x,v) = [{}^{v}/_{x}] \tag{4b}$$

By the **vars** definition, $\mathbf{vars}(x) = \{x\}$, which matches the variable in the substitution of eq. (4b).

$[\boldsymbol{p = [w_1 \mid w_2]}]$ By the **match** definition, for $v = [v_1 \mid v_2]$,

$$\mathbf{match}(p,v) = \mathbf{match}(w_1,v_1), \mathbf{match}(w_2,v_2) = [{}^{\widetilde{v_1}}/_{\widetilde{x_1}}]\,[{}^{\widetilde{v_2}}/_{\widetilde{x_2}}] \text{ where} \tag{4c}$$
$$\mathbf{match}(w_1,v_1) = [{}^{\widetilde{v_1}}/_{\widetilde{x_1}}] \tag{4d}$$
$$\mathbf{match}(w_2,v_2) = [{}^{\widetilde{v_2}}/_{\widetilde{x_2}}] \tag{4e}$$

By case analysis of w_1 and w_2 from eqs. (4d) and (4e), we conclude that

$$\mathbf{vars}(w_1) = \{\widetilde{x_1}\} \tag{4f}$$

$$\mathbf{vars}(w_2) = \{\widetilde{x_2}\} \tag{4g}$$

We need to show that $\mathbf{vars}([w_1 \mid w_2]) = \{\widetilde{x_1}, \widetilde{x_2}\}$. By the **vars** definition and eqs. (4f) and (4g), $\mathbf{vars}([w_1 \mid w_2]) = \mathbf{vars}(\widetilde{x_1}) \cup \mathbf{vars}(\widetilde{x_2}) = \{\widetilde{x_1}\} \cup \{\widetilde{x_2}\}$. This result matches the variables in the substitutions of eq. (4c).

$[\boldsymbol{p} = \{\boldsymbol{w_1}, \ldots, \boldsymbol{w_n}\}]$ Similar to the previous case. □

Lemma 7 (Closed Expression)**.** *$\mathbf{fv}(e) = \emptyset$ and $e \to e'$ implies $\mathbf{fv}(e') = \emptyset$*

Proof. By induction on the structure of e. □

Lemmata 3–7 allow us to prove Closed Term Proposition (Proposition 1). By this proposition, we can say that a closed term t remains closed, even after t transitions to some new term t', producing an action α. Lemma 7 is analogous; it states that expressions remain closed after reductions.

Proposition 1 (Closed Term)**.** *If $\mathbf{fv}(t) = \emptyset$ and $t \xrightarrow{\alpha} t'$, then $\mathbf{fv}(t') = \emptyset$*

Proof. By induction on the structure of t.

$[\boldsymbol{t} = \boldsymbol{e}]$ Holds immediately by the rule [rExpression] and the Closed Expression Lemma.

$[\boldsymbol{t} = (\boldsymbol{x} = \boldsymbol{t_1};\ \boldsymbol{t_2})]$ Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [**rLet₁**] From the rule, $t' = (x = t_1';\ t_2)$ and

$$t_1 \xrightarrow{\alpha} t_1' \tag{6a}$$

From the premise, $\mathbf{fv}(t) = \emptyset$, so by the **fv** definition, $\mathbf{fv}(t_1) \cup (\mathbf{fv}(t_2) \setminus \{x\}) = \emptyset$, or equivalently

$$\mathbf{fv}(t_1) = \emptyset \tag{6b}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{6c}$$

If we apply the inductive hypothesis to eqs. (6a) and (6b), we get

$$\mathbf{fv}(t_1') = \emptyset \tag{6d}$$

So, by eqs. (6c) and (6d) and the definition of **fv**, we get $\mathbf{fv}(x = t_1';\ t_2) = \emptyset$ as required.

2. [**rLet₂**] From the rule, $t = (x = v;\ t_2)$ and $t' = t_2\,[{}^{v}\!/\!{}_{x}]$. Since $\mathbf{fv}(t) = \emptyset$, by the Free Variables Definition, $\mathbf{fv}(v) \cup (\mathbf{fv}(t_2) \setminus \{x\}) = \emptyset$, or equivalently

$$\mathbf{fv}(v) = \emptyset \tag{6e}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{6f}$$

We need to show that $\mathbf{fv}(t') = \emptyset$, or $\mathbf{fv}(t_2\,[{}^{v}\!/\!{}_{x}]) = \emptyset$, so we consider two sub-cases:

a. If $x \notin \mathbf{fv}(t_2)$, then by Corollary 4, $t_2 = t_2\,[{}^{v}\!/\!{}_{x}]$. Substituting this in eq. (6f), results in $\mathbf{fv}(t_2\,[{}^{v}\!/\!{}_{x}]) = \emptyset$, as required.

b. If $x \in \mathbf{fv}(t_2)$, then by Lemma 5, we get $\mathbf{fv}(t_2\,[{}^{v}\!/\!{}_{x}]) = \mathbf{fv}(t_2) \setminus \{x\}$. If we substitute this in eq. (6f), the case holds.

$[\boldsymbol{t} = \mathtt{send}\,(\boldsymbol{w}, \{\mathtt{:l}, \boldsymbol{e_1},\, \ldots,\, \boldsymbol{e_n}\})]$ Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. $[\textbf{RCHOICE}_\mathbf{1}]$ From this rule, we know that $\alpha = \tau$ and

$$t' = \mathtt{send}\left(\iota, \left\{\mathtt{:l}, v_1,\, \ldots,\, v_{k-1},\, e'_k,\, \ldots,\, e_n\right\}\right)$$
$$e_k \rightarrow e'_k \tag{7a}$$

Since $\mathbf{fv}(t) = \emptyset$, then by the $\mathbf{fv}$ definition

$$\mathbf{fv}(\iota) = \emptyset \tag{7b}$$
$$\mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..k-1 \tag{7c}$$
$$\mathbf{fv}(e_i) = \emptyset \text{ for } i \in k..n \tag{7d}$$

Applying the Closed Expression Lemma to eqs. (7a) and (7d), results in $\mathbf{fv}(e_k) = \emptyset$. Using this information along with eqs. (7b–d) and the $\mathbf{fv}$ definition, results in $\mathbf{fv}(t') = \emptyset$ as required.

2. $[\textbf{RCHOICE}_\mathbf{2}]$ In this case $t = \{\mathtt{:l}, v_1,\, \ldots,\, v_n\}$ and $t' = \{\mathtt{:l}_\mu, v_1,\, \ldots,\, v_n\}$. Since from the premise $\mathbf{fv}(t) = \emptyset$, then using the $\mathbf{fv}$ definition,

$$\mathbf{fv}(\iota) = \emptyset, \quad \mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..n \tag{7e}$$

To show that $\mathbf{fv}(\{\mathtt{:l}_\mu, v_1,\, \ldots,\, v_n\}) = \emptyset$, we can apply eq. (7e) to the $\mathbf{fv}$ definition.

$[\boldsymbol{t} = \mathtt{receive\ do}\ (\{\mathtt{:l}_{\boldsymbol{i}}, \widetilde{\boldsymbol{p_i}}\} \rightarrow \boldsymbol{t_i})_{\boldsymbol{i \in I}}\mathtt{end}]$ From the premise, we know that $\mathbf{fv}(t) = \emptyset$, so by the $\mathbf{fv}$ definition,

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i}) = \emptyset \quad \text{ for all } i \in I \tag{8a}$$

Given that current structure of t, we can deduce $t \xrightarrow{\alpha} t'$ using $[\text{RBRANCH}]$, where $\alpha = ?\left\{\mathtt{:l}_j,\ v_1,\, \ldots,\, v_n\right\}$ for some $j \in I$, and

$$\mathbf{match}(\widetilde{p}_j, \widetilde{v}) = \sigma \text{ where } \sigma = [{}^{v'_1,\, \ldots,\, v'_k}\!/\!{}_{x_1,\, \ldots,\, x_k}] \tag{8b}$$
$$t' = t_j\sigma$$

From eq. (8b), we can apply Lemma 6 to get

$$\mathbf{vars}(\widetilde{p}_j) = \{x_1,\, \ldots,\, x_k\} \tag{8c}$$

Substituting eq. (8c) in eq. (8a) (for $i = j$), we get $\mathbf{fv}(t_j) \setminus \{x_1,\, \ldots,\, x_k\} = \emptyset$. Our aim is to get $t_j\sigma = \emptyset$, so we check if $x \in \mathbf{fv}(t_j)$. If this is valid, then by Lemma 5, we can conclude that $\mathbf{fv}(t_j\,[{}^{v'_1}\!/\!{}_{x_1}]) \setminus \{x_2,\, \ldots,\, x_k\} = \emptyset$. In case when $x \notin \mathbf{fv}(t_j)$, the same can be concluded by Corollary 4. Applying the same procedure for a total of k times, results in $\mathbf{fv}(t_j\,[{}^{v'_1,\, \ldots,\, v'_k}\!/\!{}_{x_1,\, \ldots,\, x_k}]) = \emptyset$, as required.

$[\boldsymbol{t} = \boldsymbol{f}\,(\boldsymbol{w}, \boldsymbol{e_2},,\, \ldots,\, \boldsymbol{e_n})]$ Given the current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. $[\textbf{RCALL}_\mathbf{1}]$ From this rule, we know that $\alpha = \tau$, $t = f\,(v_1,\, \ldots,\, v_{k-1},\, e_k,\, \ldots,\, e_n)$, $t' = f\left(v_1,\, \ldots,\, v_{k-1},\, e'_k,\, \ldots,\, e_n\right)$ and

$$e_k \rightarrow e'_k \tag{9a}$$

Since $\mathbf{fv}(t) = \emptyset$, then by the **fv** definition,

$$\mathbf{fv}(v_i) = \emptyset \quad \text{for all } i \in 1..k-1 \tag{9b}$$
$$\mathbf{fv}(e_i) = \emptyset \quad \text{for all } i \in k..n \tag{9c}$$

Applying the Closed Expression Lemma to eqs. (9a) and (9c) (for $i = k$), we get

$$\mathbf{fv}(e_k) = \emptyset \tag{9d}$$

So, using the **fv** definition with eqs. (9b–d), result $\mathbf{fv}(t') = \emptyset$ holds as expected.

2. [$\textbf{RC}\text{ALL}_\mathbf{2}$] From the rule, we know that $\alpha = f/n$ and

$$t = f\,(\iota, v_2,\, \dots,\, v_n) \tag{9e}$$
$$t' = \bar{t}\,[\iota/y]\,[v_2,\, \dots,\, v_n/x_2,\, \dots,\, x_n]$$
$$\Sigma\,(f/n) = \Omega \qquad \Omega.\texttt{body} = t \qquad \Omega.\texttt{params} = x_2,\, \dots,\, x_n \qquad \Omega.\texttt{dual} = y \tag{9f}$$

Since term reduction can only happen with respect to a well-formed *function information* environment Σ, we can assume that the only free variables in a function body are the parameter types, or formally, for all $f/n \in \mathbf{dom}(\Sigma)$, we have

$$\mathbf{fv}\big(\Sigma(f/n).\texttt{body}\big) \setminus \big(\Sigma(f/n).\texttt{params} \cup \Sigma(f/n).\texttt{dual}\big) = \emptyset$$

Thus, using this information and substituting the information from eq. (9f), we get

$$\mathbf{fv}(\bar{t}) \setminus \{y, x_2,\, \dots,\, x_n\} = \emptyset \tag{9g}$$

To obtain the expected result (*i.e.*, $\mathbf{fv}(t') = \emptyset$), we check if $y \in \mathbf{fv}(\bar{t})$. If this is true, then by Lemma 5, we can conclude that $\mathbf{fv}(\bar{t}\,[\iota/y]) \setminus \{x_2,\, \dots,\, x_n\} = \emptyset$. In case when $x \notin \mathbf{fv}(\bar{t})$, the same can be concluded by Corollary 4. Applying the same procedure for the remaining *free* variables (*i.e.*, $x_2,\, \dots,\, x_n$), we get $\mathbf{fv}(t_j\,[v'_1,\, \dots,\, v'_k/x_1,\, \dots,\, x_k]) = \emptyset$, as expected.

[$\boldsymbol{t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}}$] Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [$\textbf{RC}\text{ASE}_\mathbf{1}$] From the rule we know that $t' = \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}$, and from the premise we know that

$$e \to e' \tag{10a}$$

Since $\mathbf{fv}(t) = \emptyset$, by the **fv** definition, we know that

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i) = \emptyset \quad \text{for all } i \in I \tag{10b}$$
$$\mathbf{fv}(e) = \emptyset \tag{10c}$$

Applying Closed Expression Lemma to eqs. (10a) and (10c), results in $\mathbf{fv}(e') = \emptyset$. Thus, using this information, along with eq. (10b) and the **fv** definition, we get $\mathbf{fv}(t') = \emptyset$ as needed.

2. [$\textbf{RC}\text{ASE}_\mathbf{2}$] From the rule, we know that $t = \texttt{case}\ v'\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}$, $e = v'$ and for some $j \in I$,

$$\mathbf{match}(p_j, v') = \sigma \text{ where } \sigma = [v_1,\, \dots,\, v_n/x_1,\, \dots,\, x_n] \tag{10d}$$
$$t' = t_j\sigma \tag{10e}$$

From the premise, we know that $\mathbf{fv}(t)=\emptyset$, so by the $\mathbf{fv}$ definition, $\mathbf{fv}(v')=\emptyset$ and

$$\mathbf{fv}(t_i)\setminus\mathbf{vars}(\widetilde{p_i})=\emptyset \quad \text{for all } i\in I \tag{10f}$$

From eq. (10d), we can apply Lemma 6, to get

$$\mathbf{vars}(p_j)=\{x_1,\ \ldots,\ x_k\} \tag{10g}$$

Substituting eq. (10g) in eq. (10f) (for $i=j$), we get $\mathbf{fv}(t_j)\setminus\{x_1,\ \ldots,\ x_k\}=\emptyset$. By similar reasoning from previous cases, we get $\mathbf{fv}(t')=\emptyset$, as required. □

C.2 Proofs for Theorem 2

Before proving Theorem 2, we consider some other necessary lemmata. The Δ-Weakening Lemma weakens (*i.e.,* extends) the *session typing* environment (Δ) without affecting the overall typing result.

Lemma 8 (Δ-Weakening)**.** *If* $\Delta\cdot\Gamma\vdash^{w} S\rhd t:T\lhd S'$*, then* $(\Delta,\Delta')\cdot\Gamma\vdash^{w} S\rhd t:T\lhd S'$

Proof. Follows by induction on the derivation of $\Delta\cdot\Gamma\vdash^{w} S\rhd t:T\lhd S'$. We analyse the significant cases:

[**tRecUnknownCall**] From the rule, we know that

$$(\Delta, f/n:S)\cdot(\Gamma,\Gamma')\vdash^{y} S\rhd \bar{t}:T\lhd S' \tag{11a}$$
$$\Gamma\vdash_{\text{exp}} e_i:T_i \quad \text{for all } i\in 2..n \tag{11b}$$

Applying the inductive hypothesis to eq. (11a) results in $(\Delta,\Delta', f/n:S)\cdot(\Gamma,\Gamma')\vdash^{y} S\rhd t:T\lhd S'$, where we assume that $f/n\notin\mathbf{dom}(\Delta')$. So, using the latter result, eq. (11b) and [tRecUnknownCall] results in $(\Delta,\Delta')\cdot\Gamma\vdash^{w} S\rhd t:T\lhd S'$, as required.

[**tRecKnownCall**] From the rule, we know that

$$\Delta\,(f/n)=S \tag{12a}$$
$$\Gamma\vdash_{\text{exp}} e_i:T_i \quad \text{for all } i\in 2..n \tag{12b}$$

If we extend Δ by Δ', then $(\Delta,\Delta')(f/n)=S$ remains valid. So, using this information, along with eq. (12b) in [tRecKnownCall], we get $(\Delta,\Delta')\cdot\Gamma\vdash^{w} S\rhd t:T\lhd\mathsf{end}$, as required.

Cases [tChoice] and [tExpression] hold immediately since Δ is unused. The remaining cases hold effortlessly by the inductive hypothesis. □

The type system observes the session fidelity property if well-typed terms remain well-typed after transitioning. As terms transition, in particular in the rules [rLet$_2$], [rCall$_2$] and [rBranch], variables are substituted with values. The Substitution Lemma (Lemma 9) ensures that when free variables inside of terms and expressions are substituted with a closed value, the resulting terms and expressions remain well-typed. As a result, the substituted variables become redundant in *variable binding* environment (Γ), and thus can be removed from Γ. This lemma consists of two statements, where substitution is performed in *(i)* terms, and *(ii)* expressions.

Lemma 9 (Substitution)**.**

i. If $\Gamma\vdash_{exp} v:T'$ *and* $\Delta\cdot(\Gamma,x:T')\vdash^{w} S\rhd t:T\lhd S'$*, then* $\Delta\cdot\Gamma\vdash^{w[v/x]} S\rhd t\,[v/x]:T\lhd S'$

ii. *If* $\Gamma \vdash_{exp} v : T'$ *and* $\Gamma, x : T' \vdash_{exp} e : T$*, then* $\Gamma \vdash_{exp} e\,[{}^{v}\!/\!_{x}] : T$

Proof. By induction on the derivation of $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \rhd t : T \lhd S'$ for Item *i*, and by induction on the derivation of $\Gamma, x : T' \vdash_{\text{exp}} e : T$ for Item *ii*. We show the main cases for Item *i*:

[**tLet**] From the rule, we know that $t = (x' = t_1;\, t_2)$, and

$$x' \neq w \tag{13a}$$
$$\Gamma \vdash_{\text{exp}} v : T' \tag{13b}$$
$$\Delta \cdot (\Gamma, x : T') \vdash^{w} S \rhd t_1 : T'' \lhd S'' \tag{13c}$$
$$\Delta \cdot (\Gamma, x : T', x' : T'') \vdash^{w} S'' \rhd t_2 : T \lhd S' \tag{13d}$$

The *variable binding* environment of eq. (13d) can be reordered to

$$\Delta \cdot (\Gamma, x' : T'', x : T') \vdash^{w} S'' \rhd t_2 : T \lhd S' \tag{13e}$$

We need to show that $\Delta \cdot \Gamma \vdash^{w[{}^{v}\!/\!_{x}]} S \rhd (x' = t_1;\, t_2)\,[{}^{v}\!/\!_{x}] : T \lhd S'$, which by the Variable Substitution Definition, is equivalent to

$$\Delta \cdot \Gamma \vdash^{w[{}^{v}\!/\!_{x}]} S \rhd x' = t_1\,[{}^{v}\!/\!_{x}];\, t_2\,[{}^{v}\!/\!_{x}] : T \lhd S' \tag{13f}$$

for $x \neq x'$ and $x' \neq v$. To obtain eq. (13f), we need some preliminary results. Applying the inductive hypothesis to eqs. (13b) and (13c), and similarly to eqs. (13b) and (13e), results in

$$\Delta \cdot \Gamma \vdash^{w[{}^{v}\!/\!_{x}]} S \rhd t_1\,[{}^{v}\!/\!_{x}] : T'' \lhd S'' \tag{13g}$$
$$\Delta \cdot (\Gamma, x' : T'') \vdash^{w[{}^{v}\!/\!_{x}]} S'' \rhd t_2\,[{}^{v}\!/\!_{x}] : T \lhd S' \tag{13h}$$

From eq. (13a) and the Variable Substitution Definition we know that $x \neq w\,[{}^{v}\!/\!_{x}]$. Applying this information, along with eqs. (13g) and (13h) to the premise of [tLet] results in eq. (13f), as required.

[**tBranch**] From the rule, [tBranch], we know that for some $n \in \mathbb{N}$ and

$$\Gamma \vdash_{\text{exp}} v : T' \tag{14a}$$
$$S = \&\big\{?\mathtt{l}_i\big(T_i^1,\, \ldots,\, T_i^n\big).S_i\big\}_{i \in I}$$
$$t = \mathtt{receive\ do}\ \Big(\big\{\, \mathtt{:l}_i, p_i^1,\, \ldots,\, p_i^n \big\} \to t_i\Big)_{i \in I} \mathtt{end} \tag{14b}$$

From the premise, we also know that, for all $i \in I$:

$$\vdash^{w}_{\text{pat}} p_i^j : T_i^j \;\rhd\; \Gamma_i^j \qquad \text{for all } j \in 1..n \tag{14c}$$
$$\Delta \cdot \big(\Gamma, x : T', \Gamma_i^1,\, \ldots,\, \Gamma_i^n\big) \vdash^{w} S_i \rhd t_i : T \lhd S' \tag{14d}$$

This case holds if the following statement is obtained:

$$\Delta \cdot \Gamma \vdash^{w[{}^{v}\!/\!_{x}]} S_i \rhd t_i\,[{}^{v}\!/\!_{x}] : T \lhd S' \tag{14e}$$

where $t\,[{}^{v}\!/\!_{x}] = \mathtt{receive\ do}\ \Big(\big\{\, \mathtt{:l}_i, p_i^1,\, \ldots,\, p_i^n \big\} \to\Big)_{i \in I}\mathtt{end}$. To obtain eq. (14e) we need to use the [tBranch] rule which requires multiple premises. Applying the inductive hypothesis to eqs. (14a) and (14d) results in

$$\Delta \cdot \big(\Gamma, \Gamma_i^1,\, \ldots,\, \Gamma_i^n\big) \vdash^{w[{}^{v}\!/\!_{x}]} S_i \rhd t_i\,[{}^{v}\!/\!_{x}] : T \lhd S' \qquad \text{for all } i \in I \tag{14f}$$

If $w \neq x$, then eq. (14c)

$$\vdash^{w[v/x]}_{\text{pat}} p_i^j : T_i^j \rhd \Gamma_i^j \quad \text{for all } j \in 1..n \tag{14g}$$

since by the Variable Substitution Definition, $w = w\,[v/x]$. Therefore, eqs. (14f) and (14g) can be applied to the premise of [TBRANCH] to obtain eq. (14e):

$$\Delta \cdot \Gamma \vdash^{w[v/x]} S_i \rhd t_i\,[v/x] : T \lhd S'$$

which is the required result. In case when $w = x$, then an additional mapping may be obtained from the pattern type rule which maps the dual *pid* to some type. However, since in this case x would be substituted to a variable, then the extra mapping does not affect the result, obtaining eq. (14e) as required.

[**TCHOICE**] From the rule, we know that for some $i \in I$, $T = \{\mathtt{atom}, T_i^1, \ldots, T_i^n\}$, $S = \oplus\{!\mathtt{l}_i(\widetilde{T_i}).S_i\}_{i\in I}$ and

$$t = \mathtt{send}(\iota, \{:\mathtt{l}_i, e_1, \ldots, e_n\}) \tag{15a}$$

$$\Gamma, x : T' \vdash_{\text{exp}} e_j : T_i^j \quad \text{for all } j \in 1..n \tag{15b}$$

$$\Gamma \vdash_{\text{exp}} v : T' \tag{15c}$$

Applying eqs. (15b) and (15c) to Item *ii* of Lemma 9 results in $\Gamma \vdash_{\text{exp}} e_j\,[v/x] : T_i^j$ for all $j \in 1..n$. Applying this result to [TCHOICE] results in

$$\Delta \cdot \Gamma \vdash^{w[v/x]} S \rhd t\,[v/x] : T \lhd S'$$

which is the required result, since $t\,[v/x] = \mathtt{send}(w\,[v/x], \{:\mathtt{l}_i, e_1\,[v/x], \ldots, e_n\,[v/x]\})$. □

Lemma 10 links expression types to the basic values (and vice versa), *e.g.* the value 5 has type number.

Lemma 10 (Value Typing)**.**

i. $\Gamma \vdash_{exp} v : \mathit{boolean}$ *iff* $v = \mathit{boolean}$

ii. $\Gamma \vdash_{exp} v : \mathit{number}$ *iff* $v = \mathit{number}$

iii. $\Gamma \vdash_{exp} v : \mathit{atom}$ *iff* $v = \mathit{atom}$

iv. $\Gamma \vdash_{exp} v : \mathit{pid}$ *iff* $v = \iota$

v. $\Gamma \vdash_{exp} v : [T]$ *iff* $v = [v_1 \mid v_2]$ *or* $v = []$

vi. $\Gamma \vdash_{exp} v : \{\widetilde{T}\}$ *iff* $v = \{\widetilde{v}\}$

Proof. By case analysis on the expression typing rules. □

Lemma 11 provides a guarantee that the variables inside the substitutions produced by the **match** function have the expected types. It also ensures that the variables from the same substitutions, which are stored in Γ, are assigned with the same types. Consequently, Corollary 12 provides the same guarantees but for a *sequence* of patterns and values.

Lemma 11. *For all patterns p and values v,*

$$\left.\begin{array}{r} \textbf{\textit{match}}(p, v) = [v_1, \ldots, v_n / x_1, \ldots, x_n] \\ \vdash^{w}_{pat} p : T \rhd \Gamma \\ \emptyset \vdash_{exp} v : T \end{array}\right\} \implies \left\{\begin{array}{l} \Gamma = x_1 : T_1, \ldots, x_n : T_n \\ \emptyset \vdash_{exp} v_i : T_i \ \ \mathit{for}\ i \in 1..n \end{array}\right.$$

Proof. By induction on the definition $\textbf{match}(p,v)$. We proceed by case analysis:

$[\boldsymbol{p = b, v = b}]$ By the definition, $\textbf{match}(b,b) = [\,]$, so no substitutions are expected. By $\vdash^{w}_{\text{pat}} b : T \rhd \Gamma$ and [TPLITERAL], the *variable binding* environment (*i.e.,* Γ) must be empty, so case holds immediately.

$[\boldsymbol{p = x}]$ By definition, $\textbf{match}(x,v) = [{}^{v}\!/\!_{x}]$, and from the premise we know that

$$\emptyset \vdash_{\text{exp}} v : T. \tag{16a}$$

From $\vdash^{w}_{\text{pat}} x : T \rhd \Gamma$ and [TPVARIABLE], we know that Γ must contain $x : T$ only. Therefore, case holds by eq. (16a).

$[\boldsymbol{p = [\,w_1 \mid w_2\,], v = [\,v_1 \mid v_2\,]}]$
Using the **match** definition, $\textbf{match}([\,w_1 \mid w_2\,],[\,v_1 \mid v_2\,]) =$
$\textbf{match}(w_1,v_1), \textbf{match}(w_2,v_2)$, or equivalently

$$\textbf{match}(w_1,v_1) = [{}^{v'_1,\,\ldots,\,v'_j}\!/\!_{x_1,\,\ldots,\,x_j}] \tag{17a}$$
$$\textbf{match}(w_2,v_2) = [{}^{v'_k,\,\ldots,\,v'_n}\!/\!_{x_k,\,\ldots,\,x_n}] \text{ where } k = j+1 \tag{17b}$$

From the premise, applying [TLIST] to $\emptyset \vdash_{\text{exp}} [\,v_1 \mid v_2\,] : [\,T\,]$, results in

$$\emptyset \vdash_{\text{exp}} v_1 : T \text{ and } \emptyset \vdash_{\text{exp}} v_2 : [\,T\,] \tag{17c}$$

Applying also [TPLIST] to $\vdash^{w}_{\text{pat}} [\,w_1 \mid w_2\,] : [\,T\,] \rhd \Gamma$, results in

$$\vdash^{w}_{\text{pat}} w_1 : T \rhd \Gamma' \text{ and } \vdash^{w}_{\text{pat}} w_2 : [\,T\,] \rhd \Gamma'' \tag{17d}$$

Applying the inductive hypothesis twice to eqs. (17a–d) results in

$$\Gamma' = x_1 : T_1,\, \ldots,\, x_j : T_j \text{ and } \Gamma'' = x_k : T_k,\, \ldots,\, x_n : T_n \tag{17e}$$
$$\emptyset \vdash_{\text{exp}} v'_i : T_i \text{ for all } i \in 1..n \tag{17f}$$

Therefore, case holds by eqs. (17e) and (17f), since $\Gamma = \Gamma', \Gamma''$.

$[\boldsymbol{p = \{w_1,\, \ldots,\, w_m\}, v = \{v_1,\, \ldots,\, v_m\}}]$
Using the **match** definition, $\textbf{match}(\{w_1,\, \ldots,\, w_m\},\{v_1,\, \ldots,\, v_m\}) =$
$\textbf{match}(w_1,v_1),\, \ldots,\, \textbf{match}(w_m,v_m) = \sigma$, or equivalently, for $i \in 1..m$,

$$\textbf{match}(w_i,v_i) = \sigma_i \text{ given that } \sigma = \sigma_1,\, \ldots,\, \sigma_m \tag{18a}$$

From $\emptyset \vdash_{\text{exp}} \{v_1,\, \ldots,\, v_2\} : \{T_1,\, \ldots,\, T_m\}$, by [TTUPLE], we know that

$$\emptyset \vdash_{\text{exp}} v_i : T_i \tag{18b}$$

Applying also [TPTUPLE] to $\vdash^{w}_{\text{pat}} \{w_1,\, \ldots,\, w_m\} : \{T_1,\, \ldots,\, T_m\} \rhd \Gamma_1,\, \ldots,\, \Gamma_m$, results in

$$\vdash^{w}_{\text{pat}} w_i : T_i \rhd \Gamma_i \tag{18c}$$

Applying the inductive hypothesis m times to eqs. (18a–c) results in

$$\Gamma = \Gamma_1,\, \ldots,\, \Gamma_m = x_1 : T_1,\, \ldots,\, x_n : T_n$$
$$\emptyset \vdash_{\text{exp}} v_j : T_j \text{ for all } j \in 1..n$$

as required. □

Corollary 12. *For all patterns $\widetilde{p} = p^1, \ldots, p^n$, values $\widetilde{v} = v_1, \ldots, v_n$ and $\forall j \in 1..n$, then the following implication holds.*

$$\left.\begin{array}{r} \textit{\textbf{match}}(\widetilde{p}, \widetilde{v}) = [{}^{v'_1, \ldots, v'_k}/_{x_1, \ldots, x_k}] \\ \vdash^{y}_{pat} p^j : T^j \rhd \Gamma^j \\ \emptyset \vdash_{exp} v_j : T^j \end{array}\right\} \implies \left\{\begin{array}{l} \widetilde{\Gamma} = \Gamma^1, \ldots, \Gamma^j = x_1 : T_1, \ldots, x_k : T_k \\ \emptyset \vdash_{exp} v'_i : T_i \ \textit{for}\ i \in 1..k \end{array}\right.$$

Proof. Take $j = 1$, where we know that $\mathbf{match}(p^1, v_1) = \sigma_1$, $\vdash^{y}_{\text{pat}} p^1 : T^1 \rhd \Gamma^1$ and $\emptyset \vdash_{\text{exp}} v_1 : T^1$. Then, applying this information to Lemma 11, we get

$$\Gamma^1 = x^1_1 : T^1_1, \ldots, x^1_m : T^1_m \tag{19a}$$
$$\emptyset \vdash_{\text{exp}} v^1_i : T^1_i \text{ for } i \in 1..m \tag{19b}$$

Generalising for $j \in 1..n$, then $\widetilde{\Gamma} = \Gamma^1, \ldots, \Gamma^n$ holds by generalising eq. (19a). Also, $\emptyset \vdash_{\text{exp}} v'_i : T_i$ for $i \in 1..k$ holds by eq. (19b). Thus, Corollary 12 holds by applying Lemma 11 n times. ☐

Lemma 13 shows that the type of expressions remains unchanged (or preserved) after an expression is reduced. This means that expressions have a constant type in all steps of reductions, until the expression cannot be reduced further.

Lemma 13 (Preservation (Expressions))**.** *If $\emptyset \vdash_{exp} e : T$ and $e \rightarrow e'$, then $\emptyset \vdash_{exp} e' : T$*

Proof. Follows by induction on $\emptyset \vdash_{\text{exp}} e : T$. We consider the main cases:

[**tTuple**] From the rule, we know that $e = \{e_1, \ldots, e_k, \ldots, e_n\}$, $T = \{T_1, \ldots, T_n\}$ and

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 1..n \tag{20a}$$

Deriving $e \rightarrow e'$ using [reTuple] results in $e' = \left\{v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\right\}$ and

$$e_k \rightarrow e'_k \tag{20b}$$

Applying eqs. (20a) and (20b) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_k : T_k$. By the latter, eq. (20a) and [tTuple], we get $\emptyset \vdash_{\text{exp}} e' : T$, as required.

[**tArithmetic**] From the rule we know that $e = e_1 \diamond e_2$, $T = \texttt{number}$ and

$$\emptyset \vdash_{\text{exp}} e_1 : \texttt{number} \tag{21a}$$
$$\emptyset \vdash_{\text{exp}} e_2 : \texttt{number} \tag{21b}$$

$e \rightarrow e'$ can be derived using different rules, so we consider three sub-cases:

1. [**reOperation$_1$**] From this rule we know that $e' = e'_1 \diamond e_2$ and

$$e_1 \rightarrow e'_1 \tag{21c}$$

Applying eqs. (21a) and (21c) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_1 : \texttt{number}$. Using this information, along with eq. (21b) in [tArithmetic], results in $\emptyset \vdash_{\text{exp}} e' : \texttt{number}$, as required.

2. [**REOPERATION$_2$**] Analogous to [REOPERATION$_1$].
3. [**REOPERATION$_3$**] From the rule, we know that $e = v_1 \diamond v_2$ and e' has some value $v = v_1 \diamond v_2$. Since we know that $\emptyset \vdash_{\text{exp}} e : T$, or $\emptyset \vdash_{\text{exp}} v_1 \diamond v_2 : T$, then $\emptyset \vdash_{\text{exp}} e' : T$ follows immediately given that $e' = v = v_1 \diamond v_2$.

Regarding the remaining cases: Cases [TLITERAL], [TVARIABLE] and [TELIST] hold trivially, since $e \to e'$ does not apply. Cases [TCOMPARISON] and [TBOOLEAN] are analogous to [TARITHMETIC]. Cases [TLIST] and [TNOT] take a similar approach to [TTUPLE]. □

Lemmata 8–13 allow us to prove the Session Fidelity Theorem. This is the main result of Section 4.

Theorem 2 (Session Fidelity)**.** *If $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$ and $t \xrightarrow[\Sigma]{\alpha} t'$, then there exists some S'' and Δ', such that $\Delta' \cdot \emptyset \vdash^{w}_{\Sigma} S'' \triangleright t' : T \triangleleft S'$ for **after**$(S, \alpha) = S''$ and **after**$(\Delta, \alpha, S) = \Delta'$*

Proof. By induction on the typing derivation $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$.

[**TLET**] From the rule, we know that $x \neq w$, and

$$t = (x = t_1;\, t_2) \tag{22a}$$

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright t_1 : T' \triangleleft S''' \tag{22b}$$

$$\Delta \cdot \left(x : T'\right) \vdash^{w} S''' \triangleright t_2 : T \triangleleft S' \tag{22c}$$

From the structure of t (eq. (22a)), term transitions ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**RLET$_1$**] From this rule, we know that $t' = (x = t_1';\, t_2)$ and

$$t_1 \xrightarrow{\alpha} t_1' \tag{22d}$$

By eqs. (22b) and (22d) and the inductive hypothesis we obtain

$$\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright t_1' : T' \triangleleft S''' \tag{22e}$$

where **after**$(S, \alpha) = S''$ and **after**$(\Delta, \alpha, S) = \Delta'$. Also, by the After Function Definition, we know that Δ' is an extension of Δ, so we can apply the Δ-Weakening Lemma on eq. (22c) to get

$$\Delta' \cdot \left(x : T'\right) \vdash^{w} S''' \triangleright t_2 : T \triangleleft S' \tag{22f}$$

Using eqs. (22e) and (22f) as the premise for rule [TLET], we obtain:

$$[\text{TLET}]\ \frac{\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright t_1' : T' \triangleleft S''' \qquad \Delta' \cdot (x : T') \vdash^{w} S''' \triangleright t_2 : T \triangleleft S' \qquad x \neq w}{\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright x = t_1';\, t_2 : T \triangleleft S'}$$

where $\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright t' : T \triangleleft S'$ is the expected result.

2. [**RLET$_2$**] From the rule, we know that $t = (x = v;\, t_2)$, $t' = t_2\,[{}^{v}\!/\!_{x}]$ and $\alpha = \tau$. Since $t_1 = v$, by eq. (22b) and [TEXPRESSION], then $\emptyset \vdash_{\text{exp}} v : T'$ holds. If we apply this latter information and eq. (22c) to the Substitution Lemma, we obtain $\Delta \cdot \emptyset \vdash^{w[{}^{v}\!/\!_{x}]} S''' \triangleright t_2\,[{}^{v}\!/\!_{x}] : T \triangleleft S'$. This is the expected result, since by the Variable Substitution Definition, $w\,[{}^{v}\!/\!_{x}] = w$; and by the **after** definition, $S''' = S$ and $\Delta' = \Delta$.

[**tBranch**] From the rule, we know that for some $n \in \mathbb{N}$ and

$$S = \&\left\{?\mathtt{l}_i\left(T_i^1,\ \ldots,\ T_i^n\right).S_i\right\}_{i\in I} \tag{23a}$$

$$t = \mathtt{receive\ do}\ \left(\left\{\mathtt{:l}_i, p_i^1,\ \ldots,\ p_i^n\right\} \to t_i\right)_{i\in I}\mathtt{end} \tag{23b}$$

From the premise, we also know that some properties regarding each individual branch from the `receive` construct:

$$\forall i \in I \begin{cases} \vdash^w_{\text{pat}} p_i^j : T_i^j \ \triangleright\ \Gamma_i^j \ \text{ for all } j \in 1..n & (23c) \\ \Delta \cdot \left(\Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash^w S_i \triangleright t_i : T \triangleleft S' & (23d) \end{cases}$$

From the structure of t (eq. (23b)), term reduction ($t \xrightarrow{\alpha} t'$) can only be derived using [rBranch], where execution progresses to a single branch (*i.e.,* t_μ), rather than all branches. The right branch is chosen by matching its label, $\mathtt{l}_{i\in I}$, to the label received in the incoming message, $\mathtt{l}_\mu$. Thus, for some $k \in \mathbb{N}$, there exists some $\mu \in I$ where $\mathtt{l}_\mu = \mathtt{l}_i$, and

$$\alpha = ?\left\{\mathtt{:l}_\mu,\ v_1,\ \ldots,\ v_n\right\} \tag{23e}$$

$$\mathbf{match}((p_\mu^1,\ \ldots,\ p_\mu^n),(v_1,\ \ldots,\ v_n)) = \left[{}^{v'_1,\ \ldots,\ v'_k}/{}_{x_1,\ \ldots,\ x_k}\right] \tag{23f}$$

$$t' = t_\mu\left[{}^{v'_1,\ \ldots,\ v'_k}/{}_{x_1,\ \ldots,\ x_k}\right]$$

From eq. (23e), α refers to the message received from the dual process. We can compare the contents of this message to the original session type S (eq. (23a)), to obtain information regarding the types of the individual values inside α. We know that α contains a label $\mathtt{l}_\mu$ and n values. Thus for $j \in 1..n$, each value v_j, has a corresponding type $\boldsymbol{T_\mu^j}$ from the session type S, where S contains $?\mathtt{l}_\mu\left(\boldsymbol{T_\mu^1},\ \ldots,\ \boldsymbol{T_\mu^n}\right).S_\mu$. Formally, this can be written as

$$\emptyset \vdash_{\text{exp}} v_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{23g}$$

Applying eqs. (23c), (23f) and (23g) into Corollary 12, results in $\widetilde{\Gamma_\mu} = \Gamma_\mu^1,\ \ldots,\ \Gamma_\mu^n = x_1 : T_1,\ \ldots,\ x_k : T_k$ and

$$\emptyset \vdash_{\text{exp}} v'_m : T_m \ \text{ for } m \in 1..k \tag{23h}$$

Applying eq. (23h) and $\Delta \cdot \widetilde{\Gamma_\mu} \vdash^w S_\mu \triangleright t_\mu : T \triangleleft S'$ (from eq. (23d) for $i = \mu$) repeatedly to the Substitution Lemma, we get

$$\Delta \cdot \emptyset \vdash^w S_\mu \triangleright t_\mu\left[{}^{v'_1,\ \ldots,\ v'_k}/{}_{x_1,\ \ldots,\ x_k}\right] : T \triangleleft S' \tag{23i}$$

Since $\mathbf{after}(\&\left\{?\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i\in I}, \alpha) = S_\mu$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$, then eq. (23i) is the expected result.

[**tChoice**] From the rule, we know that for some $\mu \in I$, $T = \{\mathsf{atom}, T_\mu^1,\ \ldots,\ T_\mu^n\}$ and

$$S = \oplus\left\{!\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i\in I} \tag{24a}$$

$$t = \mathtt{send}\left(\iota, \left\{\mathtt{:l}_\mu, e_1,\ \ldots,\ e_n\right\}\right) \tag{24b}$$

$$\emptyset \vdash_{\text{exp}} e_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{24c}$$

From the structure of t (eq. (24b)), term reduction ($t \xrightarrow{\alpha} t'$) can be derived by several rules, so we have to consider two sub-cases:

1. Derived by the rule [**RCHOICE₁**], we know that $\alpha = \tau$ and

$$t' = \texttt{send}\left(\iota, \left\{ \texttt{:l}, v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n \right\}\right)$$
$$e_k \rightarrow e'_k \tag{24d}$$

Applying eq. (24c) (for $j = k$) and eq. (24d) to the Preservation (Expressions) Lemma, we get $\emptyset \vdash_{\text{exp}} e'_k : T_k$. Applying this and eq. (24c) to [TCHOICE] results in $\Delta \cdot \emptyset \vdash^{w} S \rhd t' : T \lhd S_\mu$. Since **after**$(S, \tau) = S$ and **after**$(\Delta, \alpha, S) = \Delta$, this holds.

2. [**RCHOICE₂**] From this rule we know that

$$t' = \left\{ \texttt{:l}_\mu, v_1, \ldots, v_n \right\}$$
$$\alpha = \iota \,!\left\{ \texttt{:l}_\mu, v_1, \ldots, v_n \right\} \tag{24e}$$

where α (eq. (24e)) is the message being sent to the dual process with *pid* ι.
Recall eq. (24c), where we have $\emptyset \vdash_{\text{exp}} e_j : \mathbf{T^j_-}$ for $j \in 1..n$. Notice, that the types $\boldsymbol{T^j_\mu}$ were obtained from the session type S (eq. (24a)), where S contains $!\texttt{l}_\mu\left(\boldsymbol{T^1_\mu}, \ldots, \boldsymbol{T^n_\mu} \right).S_\mu$. Now, by the premise of [RCHOICE₂], since $e_j = v_j$, then

$$\emptyset \vdash_{\text{exp}} v_j : T^j_\mu \;\text{ for all } j \in 1..n \tag{24f}$$

By the Value Typing Lemma, we also know that $\emptyset \vdash_{\text{exp}} \texttt{:l}_\mu : \texttt{atom}$. Using this latter information and eq. (24f) in [TTUPLE] and [TEXPRESSION], we get the required result:

$$\dfrac{\dfrac{\emptyset \vdash_{\text{exp}} \texttt{:l}_\mu : \texttt{atom} \qquad \forall j \in 1..n \qquad \emptyset \vdash_{\text{exp}} v_j : T^j_\mu}{\emptyset \vdash_{\text{exp}} \left\{ \texttt{:l}_\mu, v_1, \ldots, v_n \right\} : \left\{ \texttt{atom}, T^1_\mu, \ldots, T^n_\mu \right\}}\,[\text{TTUPLE}]}{\Delta \cdot \emptyset \vdash^{y} S_\mu \rhd \left\{ \texttt{:l}_\mu, v_1, \ldots, v_n \right\} : T \lhd S_\mu}\,[\text{TEXPRESSION}] \tag{24g}$$

Result from eq. (24g) holds as required, since **after**$(S, \alpha) = S_\mu$ and **after**$(\Delta, \alpha, S) = \Delta$.

[**TRECKNOWNCALL**] From the rule, we know that

$$t = f\left(w, e_2, \ldots, e_n\right) \tag{25a}$$
$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{25b}$$

From the structure of t (eq. (25a)), term transitions ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**RCALL₁**] From this rule, we know that $t = f\left(v_1, \ldots, v_{k-1}, e_k, \ldots, e_n\right)$, $\alpha = \tau$, $w = v_1$ and

$$t' = f\left(v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\right)$$
$$e_k \rightarrow e'_k \tag{25c}$$

Applying eq. (25b) (for $i = k$) and eq. (25c) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e'_k : T_k \tag{25d}$$

By eqs. (25b) and (25d) and [TRECKNOWNCALL], we get

$$\Delta \cdot \emptyset \vdash^{w} S \rhd f\left(v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\right) : T \lhd S' \tag{25e}$$

eq. (25e) holds since **after**$(S, \tau) = S$ and $v_1 = w$.

2. [**RCALL$_2$**] From the rule, we know that $\alpha = f/n$, $w = \iota$ and

$$t = f\,(\iota, v_2,\,\ldots,\,v_n) \tag{25f}$$
$$t' = \bar{t}\,[{}^{\iota}/_{y}]\,[{}^{v_2,\,\ldots,\,v_n}/_{x_2,\,\ldots,\,x_n}]$$
$$\Sigma\,(f/n) = \Omega \text{ where } \begin{cases} \Omega.\texttt{return\_type} = T \\ \Omega.\texttt{param\_types} = T_2,\,\ldots,\,T_n \end{cases} \tag{25g}$$
$$\Delta\,(f/n) = S \tag{25h}$$

Since all *known* functions (*i.e.,* $f/n \in \mathbf{dom}(\Delta)$) by eq. (25h)) are already typechecked once before, then from the *function information* environment (*i.e.,* Σ) and eq. (25g), we can assume that

$$\Delta \cdot \Gamma' \vdash^{y} S \rhd \bar{t} : T \lhd \mathsf{end} \tag{25i}$$

where Γ' contains *only* the mapping from the parameter names to their types, *i.e.,* $\Gamma' = (y : \mathsf{pid}, x_2 : T_2,\,\ldots,\,x_n : T_n)$ – our aim is to change Γ' to $\emptyset$. This assumption in eq. (25i) is possible since a well-formed Σ dictates that the only free variables in a function body are the parameter types, or formally, for all $f/n \in \mathbf{dom}(\Sigma)$, we have

$$\mathbf{fv}\big(\Sigma(f/n).\texttt{body}\big) \setminus \big(\Sigma(f/n).\texttt{params} \cup \Sigma(f/n).\texttt{dual}\big) = \emptyset$$

By eq. (25f) and Value Typing Lemma we know that $\emptyset \vdash_{\text{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (25i) to the Substitution Lemma results in

$$\Delta \cdot (x_2 : T_2,\,\ldots,\,x_n : T_n) \vdash^{y[{}^{\iota}/_{y}]} S \rhd \bar{t}\,[{}^{\iota}/_{y}] : T \lhd \mathsf{end} \tag{25j}$$

where by the Variable Substitution Definition, $y\,[{}^{\iota}/_{y}] = \iota = w$.
Applying the Substitution Lemma multiple times to eqs. (25b) and (25j), results in

$$\Delta \cdot \emptyset \vdash^{w} S \rhd \bar{t}\,[{}^{\iota}/_{y}]\,[{}^{v_2,\,\ldots,\,v_n}/_{x_2,\,\ldots,\,x_n}] : T \lhd \mathsf{end} \tag{25k}$$

as required, since $\mathbf{after}(S, f/n) = S$ and $S' = \mathsf{end}$. Also, $\mathbf{after}(\Delta, f/n, S) = (\Delta, f/n : S)$, but from eq. (25h), f/n is already mapped to S in the *session typing* environment, therefore $(\Delta, f/n : S) = \Delta$, as needed.

[**TRECUNKNOWNCALL**] From the rule, we know

$$t = f\,(w, e_2,,\,\ldots,\,e_n) \tag{26a}$$
$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{26b}$$

From the premise we also know that

$$(\Delta, f/n : S) \cdot \big(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\big) \vdash^{y} S \rhd \bar{t} : T \lhd S' \quad \text{where } \widetilde{x}, \widetilde{T}, \bar{t}, T \text{ and } y \text{ are}$$
$$\text{obtained from the } \textit{function information} \text{ environment}(\textit{i.e.,} \Sigma) \tag{26c}$$

From the structure of t (eq. (26a)), term transitions ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**RCALL$_1$**] From this rule we know that $\alpha = \tau$, and
$$t' = f\left(v_1,\, \dots,\, v_{k-1},\, e'_k,\, \dots,\, e_n\right)$$
$$e_k \to e'_k \tag{26d}$$
Applying eq. (26b) (for $i = j$) and eq. (26d) to the Preservation (Expressions) Lemma, we get
$$\emptyset \vdash_{\text{exp}} e'_j : T_j \tag{26e}$$
Using eq. (26b) and eq. (26e) in the rule [tRecUnknownCall], results in
$$\Delta \cdot \emptyset \vdash^{w} S \rhd f\left(v_1,\, \dots,\, v_{k-1},\, e'_k,\, \dots,\, e_n\right) : T \lhd S'$$
This holds since **after**$(S, \tau) = S$ and **after**$(\Delta, \tau, S) = \Delta$.
2. [**RCALL$_2$**] From the rule, we know that $\alpha = f/n$ and
$$t = f\left(\iota, v_2,\, \dots,\, v_n\right) \tag{26f}$$
$$w = \iota \tag{26g}$$
$$t' = \bar{t}\left[{}^{\iota}/_{y}\right]\left[{}^{v_2,\, \dots,\, v_n}/_{x_2,\, \dots,\, x_n}\right]$$
By eq. (26f) and the Value Typing Lemma we know that $\emptyset \vdash_{\text{exp}} \iota : \texttt{pid}$. Applying this information and eq. (26c) to the Substitution Lemma results in
$$(\Delta, f/n : S) \cdot \left(\widetilde{x} : \widetilde{T}\right) \vdash^{y[\iota/y]} S \rhd \bar{t}\left[{}^{\iota}/_{y}\right] : T \lhd S' \tag{26h}$$
where by the Variable Substitution Definition and eq. (26g), $y\left[{}^{\iota}/_{y}\right] = \iota = w$.
Applying the Substitution Lemma repeatedly to eqs. (26b) and (26h), results in
$$(\Delta, f/n : S) \cdot \emptyset \vdash^{w} S \rhd \bar{t}\left[{}^{\iota}/_{y}\right]\left[{}^{v_2,\, \dots,\, v_n}/_{x_2,\, \dots,\, x_n}\right] : T \lhd S'$$
where **after**$(S, f/n) = S$ and **after**$(\Delta, f/n, S) = (\Delta, f/n : S)$, as required.

[**TCASE**] From the rule, we know that for some type U,
$$t = \texttt{case}\ e\ \texttt{do}\ \left(p_i \to t_i\right)_{i \in I}\texttt{end} \tag{27a}$$
$$\emptyset \vdash_{\text{exp}} e : U \tag{27b}$$
$$\vdash^{w}_{\text{pat}} p_i : U \;\rhd\; \Gamma'_i \qquad \text{for all } i \in I \tag{27c}$$
$$\Delta \cdot \Gamma'_i \vdash^{w} S \rhd t_i : T \lhd S' \qquad \text{for all } i \in I \tag{27d}$$

By eq. (27a), term reduction, $t \xrightarrow{\alpha} t'$, can be derived using two rules, so we consider two sub-cases:

1. [**RCASE$_1$**] From the rule we know that $t' = \texttt{case}\ e'\ \texttt{do}\ \left(p_i \to t_i\right)_{i \in I}\texttt{end}$, and from the premise we know that
$$e \to e' \tag{27e}$$
By eqs. (27b) and (27e) and the Preservation (Expressions) Lemma, we get
$$\emptyset \vdash_{\text{exp}} e' : U \tag{27f}$$
Using eqs. (27c), (27d), (27f), and [tCase], we get
$$\Delta \cdot \emptyset \vdash^{w} S \rhd \texttt{case}\ e'\ \texttt{do}\ \left(p_i \to t_i\right)_{i \in I}\texttt{end} : T \lhd S'$$
which holds as expected since **after**$(S, \tau) = S$ and **after**$(\Delta, \tau, S) = \Delta$.

2. [$\textbf{RCASE}_\textbf{2}$] From the rule, we know that $t = \texttt{case}\ v\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}$, $e = v$ and for some $j \in I$,

$$\textbf{match}(p_j, v) = \sigma \text{ where } \sigma = [{}^{v_1,\ \ldots,\ v_n}/_{x_1,\ \ldots,\ x_n}] \tag{27g}$$

$$t' = t_j\sigma \tag{27h}$$

By eqs. (27b), (27c) and (27g) and lemma 11, we know that $\Gamma'_j = x_1 : T_1,\ \ldots,\ x_n : T_n$ and

$$\emptyset \vdash_{\text{exp}} v_k : T_k \text{ for all } k \in 1..n \tag{27i}$$

Then, by repeatedly applying the Substitution Lemma to eq. (27i), (27d for $i = j$), we get

$$\Delta \cdot \emptyset \vdash^{w} S \rhd t_j\sigma : T \lhd S'$$

This holds since $\textbf{after}(S, \tau) = S$ and $\textbf{after}(\Delta, \tau, S) = \Delta$. □

Branching pomsets for choreographies

Luc Edixhoven Sung-Shik Jongmans
Open University (Heerlen) and
CWI (Amsterdam), Netherlands
{led,ssj}@ou.nl

José Proença
CISTER, ISEP,
Polytechnic Institute of Porto, Portugal
pro@isep.ipp.pt

Guillermina Cledou
HASLab, INESC TEC & University of Minho, Portugal
mgc@inesctec.pt

Choreographic languages describe possible sequences of interactions among a set of agents. Typical denotational models are based on languages or automata over sending and receiving actions. Pomsets provide a more compact alternative by using a partial order over these actions and by not making explicit the possible interleaving of concurrent actions. However, pomsets offer no compact representation of choices. For example, if an agent Alice can send one of two possible messages to Bob three times, one would need a set of $2 \times 2 \times 2$ distinct pomsets to represent all possible branches of Alice's behaviour. This paper proposes an extension of pomsets, named *branching pomsets*, with a branching structure that can represent Alice's behaviour using $2+2+2$ ordered actions. We encode choreographies as branching pomsets and show that the pomset semantics of the encoded choreographies are bisimilar to their operational semantics.

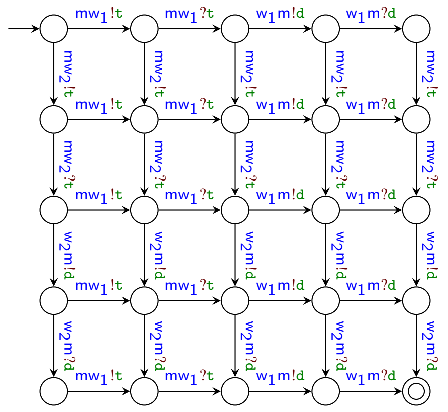
1 Introduction

Choreographic languages describe possible sequences of interactions, or communication protocols, among a set of agents. Their use is well established [9, 1, 7, 8, 2, 5], and it typically includes (1) reasoning statically over interaction properties and (2) generating code that facilitates the implementation of the concurrent protocols. Static properties include deadlock absence or the equivalence between global protocols and the parallel composition of local protocols for each agent. The code generated from choreographic languages include skeleton code for concurrent code, generated behavioural types that can be used to type-check agents, or dedicated orchestrators that dictate how the agents can interact. In this work we focus on how to analyse choreographies by proposing a new structure to compactly represent their behaviour, based on *partial-ordered multisets* (pomsets). We foresee applications of this work in both aforementioned areas.

We use two simple running examples to motivate our approach.

1. **Master-workers (MW) protocol [11].** A *master* ($\mathtt{m}$) concurrently sends *tasks* ($\mathtt{t}$) to some number of *workers* ($\mathtt{w}_1, \ldots, \mathtt{w}_n$). Once workers finish their task, they inform the master that they are *done* ($\mathtt{d}$). This protocol is expressed in our choreographic language as follows for the case of two workers.
$$(\mathtt{m}{\to}\mathtt{w}_1{:}\mathtt{t}\,;\mathtt{w}_1{\to}\mathtt{m}{:}\mathtt{d}) \;\|\; (\mathtt{m}{\to}\mathtt{w}_2{:}\mathtt{t}\,;\mathtt{w}_2{\to}\mathtt{m}{:}\mathtt{d}).$$
Here, $\mathtt{m}{\to}\mathtt{w}_1{:}\mathtt{t}$ represents an asynchronous communication from $\mathtt{m}$ to $\mathtt{w}_1$ of a message of type $\mathtt{t}$, ';' represents sequential composition and '$\|$' represents parallel composition.

Submitted to:
ICE 2022



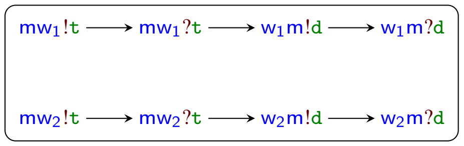


Figure 1: An automaton (left) and a pomset (right) representing the master-workers protocol.

2. **Distributed voting (DV) protocol.** Three participants – Alice (a), Bob (b) and Carol (c) – send their vote (*yes* (y) or *no* (n)) to every other participant in parallel. This is expressed as follows, where + indicates nondeterministic choice.

$$\begin{pmatrix}(a{\to}b{:}y \parallel a{\to}c{:}y)+\\(a{\to}b{:}n \parallel a{\to}c{:}n)\end{pmatrix} \parallel \begin{pmatrix}(b{\to}a{:}y \parallel b{\to}c{:}y)+\\(b{\to}a{:}n \parallel b{\to}c{:}n)\end{pmatrix} \parallel \begin{pmatrix}(c{\to}a{:}y \parallel c{\to}b{:}y)+\\(c{\to}a{:}n \parallel c{\to}b{:}n)\end{pmatrix}$$

A protocol can evolve by performing sequences of sending and receiving actions. E.g., ab!x denotes a sending action from a to b with a message of type x, and ab?x denotes the dual receiving action on b. Protocols with parallel interactions can have an explosion of states, such as our MW protocol, whose full state machine can be found on the left of Figure 1. To avoid this explosion, the state space can be represented more compactly using so-called *partially ordered multisets*, or simply pomsets [10, 6]. The right of Figure 1 shows a graphical pomset representation of the same MW protocol. The pomset contains eight events, whose labels are shown. The arrows visualise the partial order: an event precedes any other event to which it has an outgoing arrow, either directly or transitively. In this example, the event with label $mw_1!t$ precedes the event with label $mw_1?t$ directly and the events with labels $w_1m!d$ and $w_1m?d$ transitively. However, it is independent of the events involving w_2.

The behaviour represented by a pomset is the set of all its linearisations, i.e., all sequences of the labels of its events that respect their partial order. The set of linearisations of the pomset in Figure 1 consists of all interleavings of the two threads $mw_1!t\,mw_1?t\,w_1m!d\,w_1m?d$ and $mw_2!t\,mw_2?t\,w_2m!d\,w_2m?d$. This explicit concurrency yields a compact representation of the possible interleavings using just $4+4$ events, whereas the state machine needs 5×5 states to represent all interleavings. If we were to add a third worker, the automaton would grow by another factor 5, while the pomset would expand by just four additional events.

While pomsets can compactly represent concurrent behaviour, **choices** need to be represented as *sets* of pomsets: one for every branch. As a consequence, one might need an exponential number of pomsets to represent a protocol with many choices. The exponential growth is visible in our DV protocol with three participants, depicted on the left side of Figure 2. This diagram represents a set of pomsets that capture the protocol's possible behaviour, counting $2\times2\times2$ different pomsets. If we were to add a fourth participant, the set would grow by another factor 2.

This paper proposes an extension to pomsets, named *branching pomsets*, with a branching structure that can compactly represent choices. A branching pomset initially contains all branches of choices,

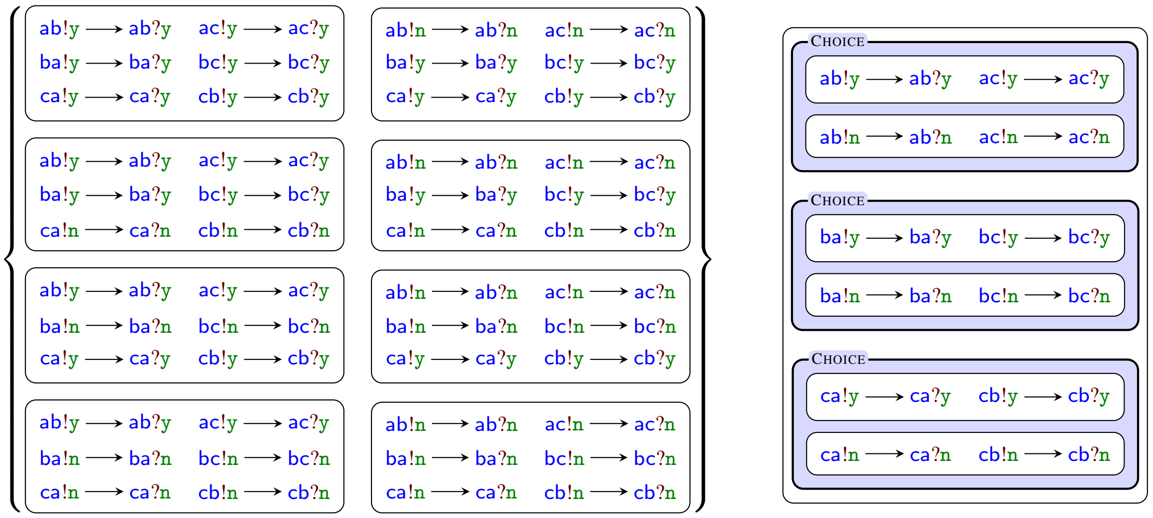


Figure 2: A set of pomsets (left) and a branching pomset (right) representing a three-participant distributed vote.

and discards non-chosen branches when firing events that require resolving a choice. The right side of Figure 2 depicts an example of a branching pomset for our DV protocol: where we would traditionally need $2 \times 2 \times 2$ pomsets (with six pairs of events each), we can represent the same behaviour as a single branching pomset with $2+2+2$ choices (with four pairs of events each). Adding an additional participant would double the number of pomsets in the set of pomsets, while it would add a single choice to the branching pomset.

To aid in the understanding of branching pomsets and their semantics, we provide a prototype tool to visualise them, available at `https://arca.di.uminho.pt/b-pomset/`. The tool provides a web interface where one can submit an input choreography, which is then visualised as a branching pomset and can be simulated. The examples and figures in the paper are already available as preset inputs. We note that the pomset simulation in our prototype currently does not support loops, for reasons which will become apparent later in the paper; however, all other operators are supported and we are most interested in (combinations of) choice and parallel composition.

Contribution This paper provides three core contributions: (1) an extension of pomsets with a branching structure, named branching pomsets, (2) an encoding from a choreographic language into branching pomsets, and (3) a formal proof that the operational semantics of a choreography and of its encoded branching pomset are equivalent, i.e., bisimilar.

Structure of the paper Section 2 presents the syntax of our choreography language and its operational semantics. Section 3 formalises branching pomsets and their semantics. Section 4 formalises how to obtain a branching pomset from a choreography and shows that a choreography and its derived branching pomset are behaviourally equivalent. Finally, Section 5 presents our conclusions and a brief discussion about future work and related work.

$$c ::= \mathbf{0} \mid \mathsf{a}{\to}\mathsf{b}{:}\mathtt{x} \mid \boxed{\mathsf{ab?x}} \mid c\,;c \mid c+c \mid c \parallel c \mid c^*$$

Figure 3: Syntax of choreographies, where a and b are participants ($\mathsf{a} \neq \mathsf{b}$) and $\mathtt{x}$ is a message type.

2 Choreographies

In this section, we formally define the syntax and semantics of our choreographic language, examples of which have been shown in the previous section.

Let $\mathcal{A}$ be the set of all participants $\mathsf{a}, \mathsf{b}, \ldots$. Let $\mathcal{L}$ be the set of actions $\{\mathsf{ab!x}, \mathsf{ab?x}\}$ for all participants $\mathsf{a} \neq \mathsf{b}$ and message types $\mathtt{x}$. For all actions the *subject* of the action is its active participant: the subject of a send action $\mathsf{ab!x}$ is a and the subject of a receive action $\mathsf{ab?x}$ is b.

The syntax is formally defined in Figure 3. Its components are standard: '$\mathbf{0}$' is the empty choreography; '$\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$' is the asynchronous communication from a to b of a message of type $\mathtt{x}$; the boxed term '$\mathsf{ab?x}$' represents a pending receive on b from a of a message of type $\mathtt{x}$, it is boxed in Figure 3 to indicate that it is only used internally to formalise behaviour but the box is not part of the syntax; '$c_1\,;c_2$', 'c_1+c_2' and '$c_1 \parallel c_2$' are respectively the weak sequential composition, nondeterministic choice and parallel composition of choreographies c_1 and c_2; finally, 'c^*' is the finite repetition (or, more informally, loop) of choreography c. The semantics for choice, parallel composition and loop are standard. We note that our sequential composition is weak. More traditionally, when sequencing c_1 and c_2, the choreography c_1 must fully terminate before proceeding to c_2. With weak sequential composition, however, actions in c_2 can already be executed as long as they do not interfere with c_1. For example, in $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,;\mathsf{c}{\to}\mathsf{d}{:}\mathtt{x}$ we can execute the action $\mathsf{cd!x}$ as it does not affect the participants of $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$: there is no dependency and thus no need to wait for $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$ to go first. However, in $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,;\mathsf{a}{\to}\mathsf{c}{:}\mathtt{x}$ the action $\mathsf{ac!x}$ cannot be executed first as its subject (a) must first execute $\mathsf{ab!x}$. This is the common interpretation of sequential composition in the context of message sequence charts [10], multiparty session types [7] and choreographic programming [2].

The reduction rules of our choreographic language are formally defined in Figure 4a and its termination rules in Figure 4b. To formalise the reduction of weak sequential composition, we follow Rensink and Wehrheim [13], who define a notion of *partial termination*.

Partial termination In a weak sequential composition $c_1\,;c_2$, an action ℓ in c_2 can be executed if c_1 can *partially terminate* for the subject of ℓ. Conceptually, a choreography c_1 can partially terminate for the subject of ℓ by discarding all branches of its behaviour which would conflict with it, i.e., in which the subject of ℓ occurs. This is written $c_1 \xrightarrow{\checkmark_\ell} c_1'$, where c_1' is the remainder of c_1 after discarding all branches involving the subject of ℓ. For example, if $c_1 = \mathsf{a}{\to}\mathsf{b}{:}\mathtt{x} + \mathsf{a}{\to}\mathsf{c}{:}\mathtt{x}$ then $c_1 \xrightarrow{\checkmark_{\mathsf{cd!x}}} \mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$, as this branch does not contain c. An exception is when the subject of ℓ occurs in *every* branch of c_1, in which case c_1 cannot partially terminate for the subject of ℓ, i.e., $c_1 \not\xrightarrow{\checkmark_\ell}$. In the above example, $c_1 \not\xrightarrow{\checkmark_{\mathsf{ad!x}}}$.

The rules for partial termination are deterministic and only discard the absolutely necessary. In the example above, $c_1 \xrightarrow{\checkmark_{\mathsf{da!x}}} c_1$ since the subject d does not occur in either branch: dropping one of the branches would be unnecessary and is thus not allowed. The rules for partial termination are defined in Figure 4c. We highlight the rules for operators:

- Sequential composition $c_1\,;c_2$ and parallel composition $c_1 \parallel c_2$ can partially terminate if both c_1 and c_2 can.
- A choice $c_1 + c_2$ can partially terminate if at least one of its branches can. If both branches can

$$
\frac{}{\mathtt{a}\to\mathtt{b}{:}\mathtt{x} \xrightarrow{\mathtt{ab!x}} \mathtt{ab?x}}
\qquad
\frac{}{\mathtt{ab?x} \xrightarrow{\mathtt{ab?x}} \mathbf{0}}
\qquad
\frac{c_1 \xrightarrow{\ell} c_1'}{c_1\,;c_2 \xrightarrow{\ell} c_1'\,;c_2}
\qquad
\frac{c_1 \xrightarrow{\checkmark_\ell} c_1' \quad c_2 \xrightarrow{\ell} c_2'}{c_1\,;c_2 \xrightarrow{\ell} c_1'\,;c_2'}
$$

$$
\frac{c_1 \xrightarrow{\ell} c_1'}{c_1 \parallel c_2 \xrightarrow{\ell} c_1' \parallel c_2}
\qquad
\frac{c_2 \xrightarrow{\ell} c_2'}{c_1 \parallel c_2 \xrightarrow{\ell} c_1 \parallel c_2'}
\qquad
\frac{c_1 \xrightarrow{\ell} c_1'}{c_1 + c_2 \xrightarrow{\ell} c_1'}
\qquad
\frac{c_2 \xrightarrow{\ell} c_2'}{c_1 + c_2 \xrightarrow{\ell} c_2'}
\qquad
\frac{c \xrightarrow{\ell} c'}{c^* \xrightarrow{\ell} c'\,;c^*}
$$

(a) Reduction rules.

$$
\frac{}{\mathbf{0}{\downarrow}}
\qquad
\frac{}{c^*{\downarrow}}
\qquad
\frac{c_1{\downarrow} \quad c_2{\downarrow} \quad \dagger \in \{;,\parallel\}}{c_1 \dagger c_2{\downarrow}}
\qquad
\frac{c_i{\downarrow} \quad i \in \{1,2\}}{c_1 + c_2{\downarrow}}
$$

(b) Termination rules.

$$
\frac{}{\mathbf{0} \xrightarrow{\checkmark_\ell} \mathbf{0}}
\qquad
\frac{c \xrightarrow{\checkmark_\ell} c}{c^* \xrightarrow{\checkmark_\ell} c^*}
\qquad
\frac{c \not\xrightarrow{\checkmark_\ell} c}{c^* \xrightarrow{\checkmark_\ell} \mathbf{0}}
\qquad
\frac{c_1 \xrightarrow{\checkmark_\ell} c_1' \quad c_2 \xrightarrow{\checkmark_\ell} c_2' \quad \dagger \in \{;,\parallel,+\}}{c_1 \dagger c_2 \xrightarrow{\checkmark_\ell} c_1' \dagger c_2'}
$$

$$
\frac{c_1 \xrightarrow{\checkmark_\ell} c_1' \quad c_2 \not\xrightarrow{\checkmark_\ell}}{c_1 + c_2 \xrightarrow{\checkmark_\ell} c_1'}
\qquad
\frac{c_1 \not\xrightarrow{\checkmark_\ell} \quad c_2 \xrightarrow{\checkmark_\ell} c_2'}{c_1 + c_2 \xrightarrow{\checkmark_\ell} c_2'}
\qquad
\frac{\mathit{subj}(\ell) \notin \{\mathtt{a},\mathtt{b}\}}{\mathtt{a}\to\mathtt{b}{:}\mathtt{x} \xrightarrow{\checkmark_\ell} \mathtt{a}\to\mathtt{b}{:}\mathtt{x}}
\qquad
\frac{\mathit{subj}(\ell) \neq \mathtt{b}}{\mathtt{ab?x} \xrightarrow{\checkmark_\ell} \mathtt{ab?x}}
$$

(c) Partial termination rules.

Figure 4: Operational semantics of choreographies.

partially terminate then both are kept, otherwise only the partially terminated one is kept.

- Following Rensink and Wehrheim, a loop c^* can partially terminate if its body (c) can partially terminate without discarding any branches, i.e., if $c \xrightarrow{\checkmark_\ell} c$. In that case also $c^* \xrightarrow{\checkmark_\ell} c^*$. Otherwise we allow c^* to be skipped entirely, represented as partial termination to $\mathbf{0}$, i.e., $c^* \xrightarrow{\checkmark_\ell} \mathbf{0}$. This can happen either if c can partially terminate to c' but $c' \neq c$, or if c cannot partially terminate at all. We use $c \not\xrightarrow{\checkmark_\ell} c$ as a shorthand to cover both these cases. Skipping a loop is necessary, for example, in a modified master-workers protocol where the master can send an arbitrary number of tasks to the workers, followed by an `end` message to indicate termination. With one worker, this protocol is expressed as $(\mathtt{m}\to\mathtt{w_1}{:}\mathtt{t}\,;\mathtt{w_1}\to\mathtt{m}{:}\mathtt{d})^*\,;\mathtt{m}\to\mathtt{w_1}{:}\mathtt{end}$. In this choreography, the loop has to eventually partially terminate to $\mathbf{0}$ to allow for the action $\mathtt{mw_1!end}$.

Example 1. Let $c_1 = (\mathtt{a}\to\mathtt{b}{:}\mathtt{x} + \mathtt{a}\to\mathtt{c}{:}\mathtt{x})\,;(\mathtt{d}\to\mathtt{b}{:}\mathtt{x} + \mathtt{d}\to\mathtt{e}{:}\mathtt{x})$. Let $c_2 = (\mathtt{a}\to\mathtt{b}{:}\mathtt{x} + \mathtt{c}\to\mathtt{b}{:}\mathtt{x})^* \parallel (\mathtt{c}\to\mathtt{a}{:}\mathtt{x} + \mathtt{c}\to\mathtt{b}{:}\mathtt{x})$.

- $c_1 \xrightarrow{\checkmark_{\mathtt{be!x}}} \mathtt{a}\to\mathtt{c}{:}\mathtt{x}\,;\mathtt{d}\to\mathtt{e}{:}\mathtt{x}$. The subject $\mathtt{b}$ of $\mathtt{be!x}$ occurs in one branch of each of both choices. While the recipient $\mathtt{e}$ also occurs in the second branch of the second choice, since it is not the actual subject it does not create a conflict.
- $c_1 \not\xrightarrow{\checkmark_{\mathtt{ab!x}}}$. While the second choice can partially terminate without reducing, the first choice contains the subject $\mathtt{a}$ of $\mathtt{ab!x}$ in both of its branches. Since one of the choices cannot reduce, neither can their sequential composition.
- $c_2 \xrightarrow{\checkmark_{\mathtt{ad!x}}} \mathbf{0} \parallel \mathtt{c}\to\mathtt{b}{:}\mathtt{x}$. The subject $\mathtt{a}$ of $\mathtt{ad!x}$ only occurs in one branch of the loop body, but the loop

can only reduce to $\mathbf{0}$. On the right hand side of the parallel composition, a occurs only in the first branch.

- $c_2 \overset{\checkmark_{\mathsf{cd!x}}}{\not\longrightarrow}$. While the loop can again reduce to $\mathbf{0}$, the subject c of cd!x occurs in both branches of the right hand side of the parallel composition. Since its right hand side cannot partially terminate, neither can it as a whole.

As already discovered by Rensink and Wehrheim [13], an unwanted consequence of these rules for partial termination is that unfolding iterations of loops no longer preserves behaviour. We would like c^* and $(c\,;c^*)+\mathbf{0}$ to behave the same, but this is not the case. For example, if $c = \mathsf{a}{\to}\mathsf{b}{:}\mathsf{x} + \mathsf{c}{\to}\mathsf{d}{:}\mathsf{x}$, then $c^* \xrightarrow{\checkmark_{\mathsf{ab!x}}} \mathbf{0}$ but $(c\,;c^*)+\mathbf{0} \xrightarrow{\checkmark_{\mathsf{ab!x}}} (\mathsf{c}{\to}\mathsf{d}{:}\mathsf{x}\,;\mathbf{0})+\mathbf{0}$. Then $c^*\,;c \xrightarrow{\mathsf{ab!x}} \mathsf{ab?x}$ by skipping the loop; however, $((c\,;c^*)+\mathbf{0})\,;c$ has no way to match this as it can skip the loop but it can only reduce the already unfolded iteration c to $\mathsf{c}{\to}\mathsf{d}{:}\mathsf{x}$ — it cannot discard it entirely. We borrow the solution that Rensink and Wehrheim offer, which is the concept *dependent guardedness*.

Dependent guardedness A loop c^* is *dependently guarded* if, for all actions ℓ, the loop body c can only partially terminate for the subject of ℓ if it does not occur in c at all. In other words: any participant that occurs in some branch of c must also occur in every other branch of c. It then follows that c can either partially terminate for the subject of ℓ without having to reduce, or it cannot partially terminate at all. Formally: if $c \xrightarrow{\checkmark_\ell} c'$ then $c' = c$. A choreography $\hat{c}$ is then dependently guarded if all of its loops are.

As a consequence, we avoid the problem above: if $c^* \xrightarrow{\checkmark_\ell} \mathbf{0}$ then $c \overset{\checkmark_\ell}{\not\longrightarrow}$ and $(c\,;c^*)+\mathbf{0}$ is also forced to reduce to the second branch of the choice, which is $\mathbf{0}$. More precisely, let c^* be some dependently guarded expression. If $c \xrightarrow{\checkmark_\ell} c'$ for some ℓ, c', then $c' = c$. It follows that $c^* \xrightarrow{\checkmark_\ell} c^*$ and $(c\,;c^*)+\mathbf{0} \xrightarrow{\checkmark_\ell} (c\,;c^*)+\mathbf{0}$. Similarly, if $c \overset{\checkmark_\ell}{\not\longrightarrow}$ then $c^* \xrightarrow{\checkmark_\ell} \mathbf{0}$ and $(c\,;c^*)+\mathbf{0} \xrightarrow{\checkmark_\ell} \mathbf{0}$.

Example 2. Let $c_1 = \mathsf{a}{\to}\mathsf{b}{:}\mathsf{x} + \mathsf{a}{\to}\mathsf{c}{:}\mathsf{x}$. Let $c_2 = \mathsf{a}{\to}\mathsf{b}{:}\mathsf{x} + \mathsf{b}{\to}\mathsf{a}{:}\mathsf{x}$.

- c_1^* is not dependently guarded as $c_1 \xrightarrow{\checkmark_{\mathsf{cd!x}}} \mathsf{a}{\to}\mathsf{b}{:}\mathsf{x} \neq c_1$. However, c_1 itself *is* dependently guarded as it does not contain any loop.

- c_2^* is dependently guarded since both a and b occur in both branches of c_2. However, $(c_2^*)^*$ is *not* dependently guarded, since $c_2^* \xrightarrow{\checkmark_{\mathsf{ab!x}}} \mathbf{0}$.

3 Branching pomsets

In this section, we formally define the syntax and semantics of branching pomsets. Additionally, we define a pomset interpretation of expressions in our choreographic language and we show this interpretation to be faithful by showing that it is bisimilar to the original choreography.

A partially ordered multiset [12], or pomset for short, consist of a set of nodes E (events), a labelling function λ to map events to some set of labels (e.g., send and receive actions), and a partial order $\leq$ to define dependencies between pairs of events (e.g., an event, or rather its corresponding action, can only fire if all events preceding it in the partial order have already fired). Its behaviour is the set of all sequences of the labels of its events that abide by $\leq$.

For example, for the pomset in Figure 1, $E = \{e_1,\dots,e_8\}$, $\lambda = \{e_1 \mapsto \mathsf{mw_1!t}, e_2 \mapsto \mathsf{mw_1?t}, e_3 \mapsto \mathsf{w_1m!d}, e_4 \mapsto \mathsf{w_1m?d}, e_5 \mapsto \mathsf{mw_2!t}, e_6 \mapsto \mathsf{mw_2?t}, e_7 \mapsto \mathsf{w_2m!d}, e_8 \mapsto \mathsf{w_2m?d}\}$, and $\leq = \{(e_i,e_j) \mid (i,j \in [1,4] \vee i,j \in [5,8]) \wedge i \leq j\}$. Its behaviour consists of all interleavings of $\mathsf{mw_1!tmw_1?tw_1m!dw_1m?d}$ and $\mathsf{mw_2!tmw_2?tw_2m!dw_2m?d}$.

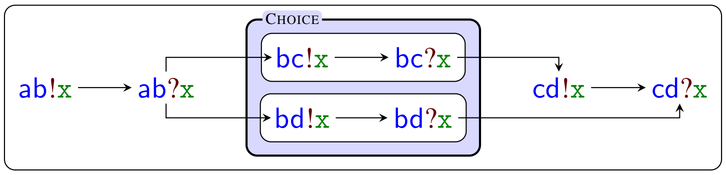


Figure 5: A branching pomset representing the choreography $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,;(\mathsf{b}{\to}\mathsf{c}{:}\mathtt{x}+\mathsf{b}{\to}\mathsf{d}{:}\mathtt{x})\,;\mathsf{c}{\to}\mathsf{d}{:}\mathtt{x}$.

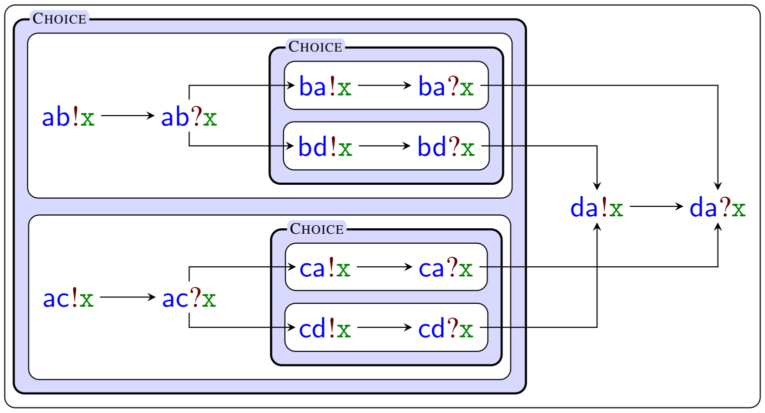


Figure 6: A branching pomset representing the choreography $((\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,;(\mathsf{b}{\to}\mathsf{a}{:}\mathtt{x}+\mathsf{b}{\to}\mathsf{d}{:}\mathtt{x}))+(\mathsf{a}{\to}\mathsf{c}{:}\mathtt{x}\,;(\mathsf{c}{\to}\mathsf{a}{:}\mathtt{x}+\mathsf{c}{\to}\mathsf{d}{:}\mathtt{x})))\,;\mathsf{d}{\to}\mathsf{a}{:}\mathtt{x}$.

As illustrated in Figure 2, however, traditional pomsets suffer from the same problem when representing choices that automata suffer from when representing concurrency: there is no explicit representation of choices in pomsets, and they are represented only implicitly by using a set of possible pomsets. We tackle this by extending pomsets with an explicit representation of choices: a branching structure on events.

Branching structure The general idea is of a branching pomset is that all possible events are initially part of the pomset, but that some are defined as being part of some choice. To fire these, all relevant choices must first be resolved by replacing the choice with one of its branches, thereby discarding the other branch. This same idea governs the operational semantics of choreographies (Figure 4): both branches of a choice are initially part of the choreography but, to proceed in one of them, the other must be dropped.

The branching structure does not interrupt the partial order and all events still participate in it, as shown in Figure 5, where arrows flow both into and out of the branches of the choice. As such, a choice can also be resolved to fire an event which is only preceded by one of the branches, reminiscent of the partial termination of choices (Figure 4c). For example, in Figure 5 the upper branch ($\mathsf{b}{\to}\mathsf{c}{:}\mathtt{x}$) can be discarded to fire the event labelled $\mathsf{cd}!\mathtt{x}$, as it is not dependent on the lower branch. As shown in Figure 6, nested choices are supported as well.

Formally, the branching structure is defined as below as a tree with root node $\mathcal{B}$, whose children are either a single event e or a choice node $\mathcal{C}$ with children (branches) $\mathcal{B}_1, \mathcal{B}_2$. All leaves are events.

$$\mathcal{B} ::= \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$$
$$\mathcal{C} ::= e \mid \{\mathcal{B}_1, \mathcal{B}_2\}$$

For example, for the pomset in Figure 5, if $E = \{e_1, \dots, e_8\}$ and $\lambda = \{e_1 \mapsto \mathtt{ab!x}, e_2 \mapsto \mathtt{ab?x}, e_3 \mapsto \mathtt{cd!x}, e_4 \mapsto \mathtt{cd?x}, e_5 \mapsto \mathtt{bc!x}, e_6 \mapsto \mathtt{bc?x}, e_7 \mapsto \mathtt{bd!x}, e_8 \mapsto \mathtt{bd?x}\}$, then its branching structure is $\{e_1, e_2, e_3, e_4, \{\{e_5, e_6\}, \{e_7, e_8\}\}\}$. For the pomset in Figure 6, if $E = \{e_1, \dots, e_{14}\}$ and $\lambda = \{e_1 \mapsto \mathtt{ab!x}, e_2 \mapsto \mathtt{ab?x}, e_3 \mapsto \mathtt{ba!x}, e_4 \mapsto \mathtt{ba?x}, e_5 \mapsto \mathtt{bd!x}, e_6 \mapsto \mathtt{bd?x}, e_7 \mapsto \mathtt{ac!x}, e_8 \mapsto \mathtt{ac?x}, e_9 \mapsto \mathtt{ca!x}, e_{10} \mapsto \mathtt{ca?x}, e_{11} \mapsto \mathtt{cd!x}, e_{12} \mapsto \mathtt{cd?x}, e_{13} \mapsto \mathtt{da!x}, e_{14} \mapsto \mathtt{da?x}\}$, then its branching structure is $\{e_{13}, e_{14}, \{\{e_1, e_2, \{\{e_3, e_4\}, \{e_5, e_6\}\}\}, \{e_7, e_8, \{\{e_9, e_{10}\}, \{e_{11}, e_{12}\}\}\}\}\}$. By resolving the outer choice and picking its upper branch ($\mathtt{a}{\rightarrow}\mathtt{b}{:}\mathtt{x}$), we drop events $e_7, \dots, e_{12}$ and obtain the middle branching pomset in Figure 8, with events $e_1, \dots, e_6, e_{13}, e_{14}$ and branching structure $\{e_1, e_2, e_{13}, e_{14}, \{\{e_3, e_4\}, \{e_5, e_6\}\}\}$.

We now formally define branching pomsets.

Definition 1 (Branching pomset)**.** A branching pomset is a four-tuple $R = \langle E, \leq, \lambda, \mathcal{B} \rangle$, where E is a set of events, $\leq \subseteq E \times E$ is such that $\leq^\star$ (the transitive closure of $\leq$) is a partial order on events, $\lambda : E \mapsto \mathcal{L}$ is a labelling function assigning an action to every event, and $\mathcal{B}$ is a branching structure such that the set of leaves of $\mathcal{B}$ is E and no event in E occurs in $\mathcal{B}$ more than once. We use $R.E$, $R.\leq$, $R.\lambda$ and $R.\mathcal{B}$ to refer to the components of R.

Semantics To fire an event in a branching pomset, on top of being minimal it must also be *active*, i.e., it must not be inside any choice. In other words: it must be a child of the branching structure's root node. We thus define a set of refinement rules in Figure 7a, written $R \sqsupseteq R'$, which can be used to resolve choices and move events upwards in the branching structure.

The first two rules, REFL and TRANS, are straightforward. The third rule, CHOICE, resolves choices. It states that we can replace a choice with one of its branches. This rule serves a dual purpose: by applying it to the outer choice of the pomset in Figure 6 we can fire the event $\mathtt{ab!x}$ in its first branch; alternatively, by applying it to the pomset in Figure 5 we can discard one branch of the choice and then fire the event $\mathtt{cd!x}$, which is now minimal. The latter use corresponds with the partial termination rules for choreographies. The fourth rule, CONGR is used for more fine-grained partial termination. To make the event $\mathtt{da!x}$ minimal in Figure 6 we could resolve two choices with CHOICE (and TRANS). However, as the rules for partial termination tell us, it is unnecessary to resolve the outer choice. Instead, we can apply CHOICE to both inner choices and apply CONGR to the outer choice to update it without unnecessarily resolving it. Finally, the fifth rule overloads the refinement notation to also apply to branching pomsets themselves: if $R.\mathcal{B}$ can refine to some $\mathcal{B}'$, then R itself can refine to a derived branching pomset with branching structure $\mathcal{B}'$, whose events are restricted to those occurring in $\mathcal{B}'$ and likewise for $\leq$ and λ.

The reduction and termination rules are defined in Figure 7b. The first rule simply states that a pomset can terminate if its branching structure can reduce to the empty set. The second rule defines the conditions for *enabling* an event e, written $R \xrightarrow{\checkmark_e} R'$. A branching pomset R can enable e by refining to R' if e is both minimal and active in R' ($e \in \text{a-min}(R')$), and if there is no other refinement in between in which e is already minimal and active. In other words, R may only refine as far as strictly necessary to enable e. This rule implements the same idea as partial termination, with the subtle difference that, whereas partial termination tries to remove any occurrence of a participant, in this case e is actually an event in R itself. As the two notions are very similar, we use the same notation for enabling events in branching pomsets as for partial termination. Finally, the last two rules state that, if R can enable e by refining to R', then it can fire e by reducing to $R' - e$, which is the branching pomset obtained by removing e from R' (Figure 7c). This reduction is defined both on e's label and on the event itself, the latter for internal use in proofs since $\lambda(e)$ is typically not unique but e is.

Example 3.

$$\frac{}{\mathcal{B} \sqsupseteq \mathcal{B}}[\textsc{Refl}] \quad \frac{\mathcal{B} \sqsupseteq \mathcal{B}' \sqsupseteq \mathcal{B}''}{\mathcal{B} \sqsupseteq \mathcal{B}''}[\textsc{Trans}] \quad \frac{i \in \{1,2\}}{\{\{\mathcal{B}_1,\mathcal{B}_2\}\} \cup \mathcal{B} \sqsupseteq \mathcal{B}_i \cup \mathcal{B}}[\textsc{Choice}]$$

$$\frac{\mathcal{B}_1 \sqsupseteq \mathcal{B}_1' \quad \mathcal{B}_2 \sqsupseteq \mathcal{B}_2'}{\{\{\mathcal{B}_1,\mathcal{B}_2\}\} \cup \mathcal{B} \sqsupseteq \{\{\mathcal{B}_1',\mathcal{B}_2'\}\} \cup \mathcal{B}}[\textsc{Congr}] \quad \frac{R.\mathcal{B} \sqsupseteq \mathcal{B}'}{R \sqsupseteq R[\mathcal{B}']}$$

(a) Refinement rules, where we assume for CHOICE and CONGR that $\{\mathcal{B}_1,\mathcal{B}_2\} \notin \mathcal{B}$.

$$\frac{R.\mathcal{B} \sqsupseteq \emptyset}{R\downarrow} \quad \frac{\begin{array}{c} R \sqsupseteq R' \quad e \in \text{a-min}(R') \\ \forall R'' : R \sqsupseteq R'' \sqsupset R' \Rightarrow e \notin \text{a-min}(R'') \end{array}}{R \xrightarrow{\checkmark_e} R'} \quad \frac{R \xrightarrow{\checkmark_e} R'}{R \xrightarrow{e} R' - e} \quad \frac{R \xrightarrow{e} R'}{R \xrightarrow{\lambda(e)} R'}$$

(b) Reduction and termination rules.

$$\begin{aligned}
\langle E, \leq, \lambda, \mathcal{B}\rangle[\mathcal{B}'] &= \langle E|_{\mathcal{B}'}, \leq|_{\mathcal{B}'}, \lambda|_{\mathcal{B}'}, \mathcal{B}'\rangle \\
X|_{\mathcal{B}} &= \text{restricts } X \text{ only to the events in } \mathcal{B} \\
\text{a-min}(R) &= \{e \in R.E \mid \nexists e' \in R.E : e' < e\} \wedge e \in R.\mathcal{B} \\
\hat{e} - e &= \hat{e} \\
\{\mathcal{C}_1, \ldots, \mathcal{C}_n\} - e &= \begin{cases} \{\mathcal{C}_1, \ldots, \mathcal{C}_{i-1}, \mathcal{C}_{i+1}, \ldots, \mathcal{C}_n\} & \text{if } \mathcal{C}_i = e \\ \{\mathcal{C}_1 - e, \ldots, \mathcal{C}_n - e\} & \text{otherwise} \end{cases} \\
\{\mathcal{B}_1, \mathcal{B}_2\} - e &= \{\mathcal{B}_1 - e, \mathcal{B}_2 - e\} \\
R - e &= R[R.\mathcal{B} - e]
\end{aligned}$$

(c) Operations on branching pomsets.

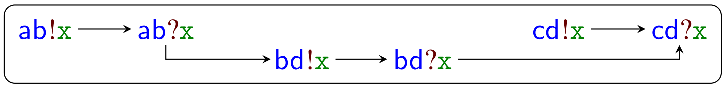
Figure 7: Semantics of branching pomsets.

- $R \xrightarrow{\checkmark_e} R'$, where R is the branching pomset in Figure 5, R' is the topmost branching pomset in Figure 8 and e is the event with label `cd!x`.
- $R \xrightarrow{\checkmark_e} R'$, where R is the branching pomset in Figure 6, R' is the middle branching pomset in Figure 8 and e is the event with label `ab!x`.
- $R \xrightarrow{\checkmark_e} R'$, where R is the branching pomset in Figure 6, R' is the middle branching pomset in Figure 8 and e is the event with label `da!x`.

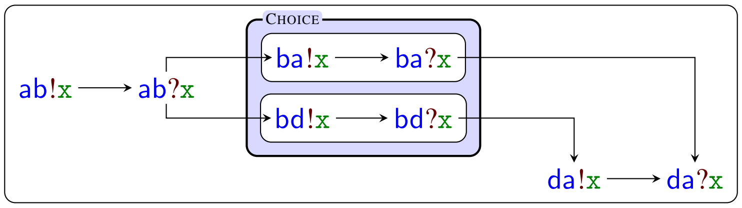
4 Branching pomsets for choreographies

In this section we formalise the construction of a branching pomset for a choreography c and we show that the pomset semantics for the branching pomset are bisimilar to the operational semantics for c.

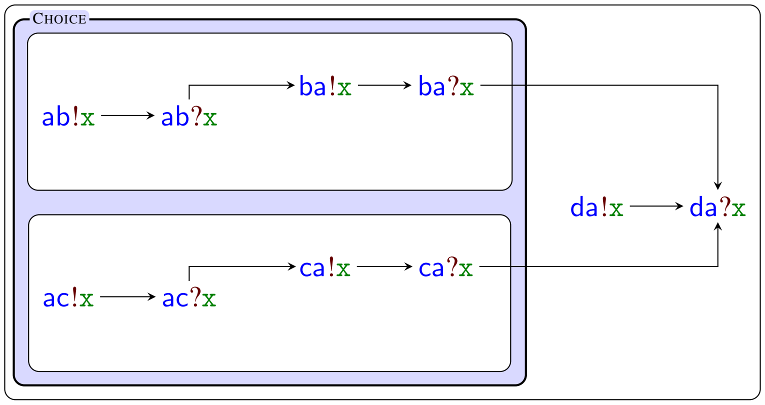
We have given examples of choreographies and corresponding branching pomsets in Figures 5 and 6. Formally, the rules for the construction of a branching pomset for a choreography c, written $[\![c]\!]$, are defined in Figure 9. Most rules are as expected. We highlight the rules for operators.



Obtained by applying CHOICE to the pomset in Figure 5.



Obtained by applying CHOICE to the outer choice of the pomset in Figure 6.



Obtained by applying CONGR to the outer choice and CHOICE to both inner choices of the pomset in Figure 6.

Figure 8: Three refined pomsets.

- The rule for parallel composition ($[\![c_1 \parallel c_2]\!]$) takes the pairwise union of all components.
- The rule for sequential composition ($[\![c_1 \,;c_2]\!]$) also adds dependencies to ensure that, for every a, all events with subject a in $[\![c_1]\!]$ (denoted E_{1_a}) must precede all events with subject a in $[\![c_2]\!]$. This matches the reduction rule for weak sequential composition of choreographies (Figure 4a), as events in $[\![c_2]\!]$ are only required to wait for events in $[\![c_1]\!]$ whose subject is the same.
- The rule for choice ($[\![c_1 + c_2]\!]$) adds a single top-level choice in the branching structure to choose between the pomsets for c_1 and c_2.
- The rule for loops ($[\![c^*]\!]$) encodes a loop as a choice between terminating ($\mathbf{0}$) and unfolding one iteration of the loop ($c\,;c^*$). This results in a pomset of infinite size. We note that our theoretical results still hold even on infinite pomsets, but that any analysis of an infinite pomset will have to be symbolic. However, since the focus of this paper is on supporting choices, we do not discuss this further and leave symbolic analyses for loops for future work.

As an example, we construct part of the branching pomset in Figure 5: $(\mathsf{b}{\to}\mathsf{c}{:}\mathsf{x} + \mathsf{b}{\to}\mathsf{d}{:}\mathsf{x})\,;\mathsf{c}{\to}\mathsf{d}{:}\mathsf{x}$ (thus omitting $\mathsf{a}{\to}\mathsf{b}{:}\mathsf{x}$). Let $[\![\mathsf{b}{\to}\mathsf{c}{:}\mathsf{x}]\!] = \langle\{e_1,e_2\},\leq_1,\lambda_1,\{e_1,e_2\}\rangle$, $[\![\mathsf{b}{\to}\mathsf{d}{:}\mathsf{x}]\!] = \langle\{e_3,e_4\},\leq_2,\lambda_2,\{e_3,e_4\}\rangle$ and $[\![\mathsf{c}{\to}\mathsf{d}{:}\mathsf{x}]\!] = \langle\{e_5,e_6\},\leq_3,\lambda_3,\{e_5,e_6\}\rangle$ as in Figure 9. First, $[\![\mathsf{b}{\to}\mathsf{c}{:}\mathsf{x} + \mathsf{b}{\to}\mathsf{d}{:}\mathsf{x}]\!] = \langle\{e_1,\ldots,e_4\},\leq_1\cup\leq_2,\lambda_1\cup\lambda_2,\{\{\{e_1,e_2\},\{e_3,e_4\}\}\}\rangle$; this is the pairwise union of the first three components, with the

$$\begin{aligned}
[\![\mathbf{0}]\!] &= \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \\
[\![\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}]\!] &= \langle \{e_1, e_2\}, \{e_1 \leq e_1, e_1 \leq e_2, e_2 \leq e_2\}, \{e_1 \mapsto \mathsf{ab}!\mathtt{x}, e_2 \mapsto \mathsf{ab}?\mathtt{x}\}, \{e_1, e_2\} \rangle \\
[\![\mathsf{ab}?\mathtt{x}]\!] &= \langle \{e\}, \{e \leq e\}, \{e \mapsto \mathsf{ab}?\mathtt{x}\}, \{e\} \rangle \\
[\![c_1 \dagger c_2]\!] &= [\![c_1]\!] \dagger [\![c_2]\!] \text{ for } \dagger \in \{;, +, \|\} \\
[\![c^*]\!] &= [\![(c\,;c^*) + \mathbf{0}]\!]
\end{aligned}$$

$$\begin{aligned}
R_1\,; R_2 &= \langle E_1 \cup E_2, \leq_1 \cup \leq_2 \cup \textstyle\bigcup_{\mathsf{a} \in \mathcal{A}} E_{1_\mathsf{a}} \times E_{2_\mathsf{a}}, \lambda_1 \cup \lambda_2, \mathcal{B}_1 \cup \mathcal{B}_2 \rangle \\
R_1 + R_2 &= \langle E_1 \cup E_2, \leq_1 \cup \leq_2, \lambda_1 \cup \lambda_2, \{\{\mathcal{B}_1, \mathcal{B}_2\}\} \rangle \\
R_1 \parallel R_2 &= \langle E_1 \cup E_2, \leq_1 \cup \leq_2, \lambda_1 \cup \lambda_2, \mathcal{B}_1 \cup \mathcal{B}_2 \rangle
\end{aligned}$$

Figure 9: Pomset interpretation of choreographies, where $R_i = \langle E_i, \leq_i, \lambda_i, \mathcal{B}_i \rangle$ for $i \in \{1,2\}$, $\mathcal{A}$ is the set of all participants $(\mathsf{a}, \mathsf{b}, \ldots)$ and E_{i_a} is the subset of events in E_i with subject a.

branching structure adding a choice between the two branches. Then $[\![(\mathsf{b}{\to}\mathsf{c}{:}\mathtt{x} + \mathsf{b}{\to}\mathsf{d}{:}\mathtt{x})\,; \mathsf{c}{\to}\mathsf{d}{:}\mathtt{x}]\!] = \langle \{e_1, \ldots, e_6\}, \leq_1 \cup \leq_2 \cup \leq_3 \cup \{e_2 \leq e_5, e_4 \leq e_6\}, \lambda_1 \cup \lambda_2 \cup \lambda_2, \{e_5, e_6, \{\{e_1, e_2\}, \{e_3, e_4\}\}\} \rangle$; again, this is the pairwise union of all components, with the addition of two dependencies: $e_2 \leq e_5$ represents the arrow in Figure 5 from $\mathsf{bc}?\mathtt{x}$ to $\mathsf{cd}!\mathtt{x}$ as they both have subject c, $e_4 \leq e_6$ represents the arrow from $\mathsf{bd}?\mathtt{x}$ to $\mathsf{cd}?\mathtt{x}$ as they both have subject d. There are no direct dependencies between e_1 ($\mathsf{bc}!\mathtt{x}$) or e_3 ($\mathsf{bd}!\mathtt{x}$) and either e_5 or e_6, as the latter two do not have subject b.

Bisimulation For any given a choreography c we can derive two labelled transition systems: one from the operational semantics in Figure 4 over c, and one from the pomset semantics in Figure 7 over the branching pomset $[\![c]\!]$ produced by the rules in Figure 9. In the remainder of this section we show that the two transition systems are bisimilar.

Two systems are language equivalent (or trace equivalent) if their languages are the same, i.e., if they accept the same set of words (or traces). Two systems are bisimilar if each of them can simulate the other, i.e., if they cannot be distinguished from each other just by looking at their behaviour. This is a stronger notion of equivalence than language equivalence: if two systems are bisimilar then they are also language equivalent, but the inverse is not necessarily true.

Example 4.

- $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,; (\mathsf{b}{\to}\mathsf{a}{:}\mathtt{x} + \mathsf{b}{\to}\mathsf{a}{:}\mathtt{y})$ is language equivalent but not bisimilar to $(\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,; \mathsf{b}{\to}\mathsf{a}{:}\mathtt{x}) + (\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}\,; \mathsf{b}{\to}\mathsf{a}{:}\mathtt{y})$. In the former the choice between $\mathsf{b}{\to}\mathsf{a}{:}\mathtt{x}$ and $\mathsf{b}{\to}\mathsf{a}{:}\mathtt{y}$ is made only after $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$, while in the latter the choice is made up front. As a result, it is possible in the latter system to fire $\mathsf{ab}!\mathtt{x}\,; \mathsf{ab}?\mathtt{x}$ and then end up in a state where $\mathsf{ba}!\mathtt{x}$ cannot be fired because the branch with $\mathsf{b}{\to}\mathsf{a}{:}\mathtt{y}$ was chosen — or the other way around; in the former system it is always possible to fire both $\mathsf{ba}!\mathtt{x}$ and $\mathsf{ba}!\mathtt{y}$.
- $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$ is bisimilar to $\mathsf{a}{\to}\mathsf{b}{:}\mathtt{x} + \mathsf{a}{\to}\mathsf{b}{:}\mathtt{x}$. While the latter contains a choice, the two systems cannot be distinguished by their behaviour. In both cases, the only allowed action is $\mathsf{ab}!\mathtt{x}$ and then $\mathsf{ab}?\mathtt{x}$.

Formally, two transition systems A_1, A_2 are bisimilar, written $A_1 \sim A_2$, if there exists a bisimulation relation $\mathcal{R}$ relating the states of A_1 and A_2 which relates their initial states [14]. The relation $\mathcal{R}$ is a bisimulation relation if, for every pair of states $\langle p, q \rangle \in \mathcal{R}$:

- If $p \xrightarrow{\ell} p'$ then $q \xrightarrow{\ell} q'$ and $\langle p', q' \rangle \in \mathcal{R}$ for some q', and vice-versa.

- If $p{\downarrow}$ then $q{\downarrow}$, and vice-versa.

In other words: if one of the two can perform a step, then the other can perform a matching step such that the resulting states are again in the bisimulation relation.

This is also the approach we follow when proving that $c \sim [\![c]\!]$ for all (dependently guarded) choreographies c: we define a relation $\mathcal{R} = \{\langle c, [\![c]\!]\rangle \mid c \text{ is a dependently guarded choreography}\}$ relating all dependently guarded choreographies with their interpretation as branching pomset by the rules in Figure 9. We then show that:

- If $c \xrightarrow{\ell} c'$ then $[\![c]\!] \xrightarrow{\ell} [\![c']\!]$ (Lemma 2).
- If $[\![c]\!] \xrightarrow{\ell} R'$ then $c \xrightarrow{\ell} c'$ such that $R' = [\![c']\!]$ (Lemma 3).
- If $c{\downarrow}$ then $[\![c]\!]{\downarrow}$ (Lemma 4).
- If $[\![c]\!]{\downarrow}$ then $c{\downarrow}$ (Lemma 5).

Together these lemmas prove that $c \sim [\![c]\!]$ for all dependently guarded c (Theorem 6). Most of the proofs are straightforward by structural induction on c. Of particular interest, however, are the two reduction lemmas in the case of weak sequential composition, i.e., if $c_1 \mathbin{;} c_2 \xrightarrow{\ell} c_1' \mathbin{;} c_2'$ in Lemma 2 and if $[\![c_1 \mathbin{;} c_2]\!] \xrightarrow{e} R'$ where e is an event in $[\![c_2]\!]$ in Lemma 3. To prove these specific cases we need to show a correspondence between partial termination and enabling events. We do this with Lemma 1, in which we show two directions simultaneously. If the choreography c_1 can partially terminate for the subject of an action ℓ in c_2 then the branching pomset $[\![c_1 \mathbin{;} c_2]\!]$ can enable the corresponding event. Conversely, if $[\![c_1 \mathbin{;} c_2]\!]$ can enable some event in $[\![c_2]\!]$ then the choreography c_1 can partially terminate for the subject of its label. When proving these cases in Lemmas 2 and 3, we then only have to show that the preconditions of Lemma 1 hold.

In the following, a number of technical lemmas and most of the proofs are omitted in favour of informal proof sketches or highlights. The omitted proofs can be found in Appendix A, the omitted technical lemmas in Appendix B.

Lemma 1. *Let c_1 and c_2 be dependently guarded choreographies. Let $c_2 \xrightarrow{\ell} c_2'$ and $[\![c_2]\!] \xrightarrow{\checkmark_e} R_2'$ such that $\lambda(e) = \ell$ and $[\![c_2']\!] = R_2' - e$.*

(a) If $c_1 \xrightarrow{\checkmark_\ell} c_1'$ then $[\![c_1 \mathbin{;} c_2]\!] \xrightarrow{\checkmark_e} [\![c_1']\!] \mathbin{;} R_2'$.

(b) If $[\![c_1 \mathbin{;} c_2]\!] \xrightarrow{\checkmark_e} R_1' \mathbin{;} R_2'$ then $c_1 \xrightarrow{\checkmark_{\lambda(e)}} c_1'$ and $[\![c_1']\!] = R_1'$.

Proof sketch. This proof is by structural induction on c_1. Although the details require careful consideration, it is conceptually straightforward: every case in (a) consists of showing that e is minimal and active in $[\![c_1']\!] \mathbin{;} R_2'$ and that $[\![c_1']\!] \mathbin{;} R_2'$ is the first refinement for which this is true, and then applying the second rule in Figure 7b; every case in (b) consists of showing that $[\![c_3 \mathbin{;} c_2]\!] \xrightarrow{\checkmark_e} [\![c_3']\!] \mathbin{;} R_2'$ for some subexpression c_3 of c_1 and similarly for c_4 (e.g., when $c_1 = c_3 + c_4$), then applying the induction hypothesis (b) to obtain $c_3 \xrightarrow{\checkmark_\ell} c_3'$ and $c_4 \xrightarrow{\checkmark_\ell} c_4'$, and finally applying the partial termination rules in Figure 4c. □

Lemma 2. *Let c be a dependently guarded choreography. If $c \xrightarrow{\ell} c'$ then $[\![c]\!] \xrightarrow{\ell} [\![c']\!]$.*

Proof sketch. This proof is by structural induction on c. We note that, if $c = c_1 \mathbin{;} c_2$ and $c' = c_1' \mathbin{;} c_2'$, i.e., when partial termination is applied, then the premises of Lemma 1 hold by the induction hypothesis and the result swiftly follows. All other cases are straightforward. □

Lemma 3. *Let c be a dependently guarded choreography. If $[\![c]\!] \xrightarrow{\ell} R'$ for some R' then $c \xrightarrow{\ell} c'$ such that $R' = [\![c']\!]$.*

Proof sketch. This proof is by structural induction on c. We highlight two cases:

- If $c = c_1^*$ then we use a technical lemma to show that $R' = R'_1 \,;\, [\![c_1^*]\!]$ such that $[\![c_1]\!] \xrightarrow{\ell} R'_1$. It then follows from the induction hypothesis that $c_1 \xrightarrow{\ell} c'_1$ such that $[\![c'_1]\!] = R'_1$. The remainder is straightforward.

- If $c = c_1 \,;\, c_2$ then $[\![c]\!] = [\![c_1]\!] \,;\, [\![c_2]\!]$. If e is an event in $[\![c_2]\!]$ then we proceed to show that $[\![c_2]\!] \xrightarrow{\ell} R'_2$, at which point we can apply the induction hypothesis. We have then satisfied the premises of Lemma 1. The remainder is straightforward.

All other cases are straightforward. □

Lemma 4. *Let c be a dependently guarded choreography. If $c\downarrow$ then $[\![c]\!]\downarrow$.*

Proof sketch. This proof is by structural induction on c. All cases are straightforward. □

Lemma 5. *Let c be a dependently guarded choreography. If $[\![c]\!]\downarrow$ then $c\downarrow$.*

Proof sketch. This proof is by structural induction on c. All cases are straightforward. □

Theorem 6. *Let c be a dependently guarded choreography. Then $c \sim [\![c]\!]$.*

Proof. Recall the relation $\mathcal{R} = \{\langle c, [\![c]\!]\rangle \mid c$ is a dependently guarded choreography$\}$. Let $\langle c, R\rangle \in \mathcal{R}$.

- If $c \xrightarrow{\ell} c'$ then $R \xrightarrow{\ell} R'$ and $\langle c', R'\rangle \in \mathcal{R}$ (Lemma 2).
- If $R \xrightarrow{\ell} R'$ then $c \xrightarrow{\ell} c'$ and $\langle c', R'\rangle \in \mathcal{R}$ (Lemma 3).
- If $c\downarrow$ then $R\downarrow$ (Lemma 4).
- If $R\downarrow$ then $c\downarrow$ (Lemma 5).

Then $\mathcal{R}$ is a bisimulation relation and $c \sim [\![c]\!]$ ([14]). □

5 Conclusion

We have defined a choreography language and its operational semantics (Figures 3 and 4) using the weak sequential composition and partial termination of Rensink and Wehrheim [13], which is novel in the context of choreographies. We have defined a model, branching pomsets (Definition 1), which can compactly represent both concurrency and choices, and have defined its semantics (Figure 7). We have shown that we can use branching pomsets to model choreographies (Figure 9) and that this model is behaviourally equivalent to the operational semantics (Theorem 6).

We believe that branching pomsets can be further improved. We mention three points in particular and then discuss related work.

Binary choices Our branching structure $\mathcal{B}$ only supports binary choices. This matches the structure of choreographies, but it would be more natural to represent $c_1 + (c_2 + c_3)$ as a single choice between the pomsets $[\![c_1]\!]$, $[\![c_2]\!]$ and $[\![c_3]\!]$ instead of as two nested binary choices. However, supporting arbitrary n-ary choices also requires some thought about how to change the rules for refinement (Figure 7a), in particular CHOICE. A naive change would be to simply have this rule use $i \in \{1, \ldots, n\}$ and $\{\{\mathcal{B}_1, \ldots, \mathcal{B}_n\}\}$ instead of its current binary rules, but this is not sufficient as this naive n-ary choice would not be equivalent to the same branches composed as nested binary choices. For example, $c_1 + (c_2 + c_3)$ can partially terminate to $c_1 + c_2$ and its interpretation as a branching pomset can refine to $[\![c_1 + c_2]\!]$, but a branching pomset whose branching structure consists of a single ternary choice $\{\{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3\}\}$ would not be able to refine to $\{\{\mathcal{B}_1, \mathcal{B}_2\}\}$ as the rules would only allow it to refine all of its branches or discard all but one of them. Properly supporting n-ary choices would thus also require a new rule that allows $\{\{\mathcal{B}_1, \ldots, \mathcal{B}_m\}\}$ to refine to choice between an arbitrary (non-empty) subset of its branches.

Partial order In Definition 1, $\leq$ is defined as a relation on events such that its transitive closure is a partial order, rather than $\leq$ being a partial order itself as it is in traditional pomsets. The need for this change arises from the update rule $R[\mathcal{B}]$ (Figure 7c) in our use case as choreographies. Consider the branching pomset in Figure 5. To match the operational semantics, we should be able to refine this pomset by discarding the b→c:x branch of the choice, after which cd!x should be minimal. In our current rules the events bc!x and bc?x are removed along with their entries in $\leq$ and then cd!x is minimal. However, if $\leq$ is a partial order, then since a partial order is transitive $\leq$ would also contain the entries ab!x $\leq$ cd!x and ab?x $\leq$ cd!x and, since these entries do not contain bc!x or bc?x but are obtained by transitivity, they are not removed. Consequently, there would be no refinement that enables cd!x.

In general, if $R_1 \sqsupseteq R_1'$ and $R_2 \sqsupseteq R_2'$ then it would not necessarily be true that $R_1 ; R_2 \sqsupseteq R_1' ; R_2'$, as $R_1 ; R_2$ may contain dependencies obtained by transitivity which would still be present in its updated version but which cannot be derived in $R_1' ; R_2'$. We have no ready alternative. In the case of choreographies it may suffice to provide a more sophisticated update rule which properly trims these unwanted dependencies, but since this relies on knowledge of how these dependencies were derived from choreographies it is difficult to see how this could be applied to branching pomsets in general.

Loops In Figure 9 a loop c^* is encoded by infinitely unfolding it. As such, branching pomsets do not currently provide a finite representation of infinite choreographies. This remains a topic for future work, for which we envision two possible directions. One possibility would be to add an explicit repetition construct to the branching structure (e.g., change the second grammatical rule to $\mathcal{C} = e \mid \{\mathcal{B}_1, \mathcal{B}_2\} \mid \mathcal{B}^*$) and expand the semantics and proofs accordingly. Another possibility would be to explore the approach used in message sequence chart graphs [1] and add a graph structure on top of the branching structure.

Related work Choreographies are typically used in a top-down workflow: the developer writes a global view C and decomposes it into its projections, such that the behaviour of C is *behaviourally equivalent* to the parallel composition of its projections. Examples of this approach include workflows based on message sequence charts [9, 1], multiparty session types [7, 8], and choreographic programs [2, 5]. The choreographic language used in this paper assumes asynchronous communication between agents and includes a finite loop operator, borrowing from this literature the same notion of actions as interactions and their (parallel, sequential, and choice) composition.

Pomsets were initially introduced by Pratt [12] for concurrent models and have been widely used, e.g., in the context of message sequence charts by Katoen and Lambert [10]. Recently Guanciale and

Tuosto proposed two semantic frameworks for choreographies, one of which uses sets of pomsets [15]. They also note that the pomset framework exhibits exponential growth in the number of choices in a choreography, and they propose an alternative semantic framework using hypergraphs, which can compactly represent choices. While the hypergraph framework is more compact, their pomset framework is simpler and, they believe, more elegant. We agree with this analysis, and we aim to preserve the simplicity and elegance of the pomset framework by proposing a semantic framework that avoids exponential growth in the number of choices while still being based on pomsets. In another recent paper they use pomsets to reason over choreography realisability [6]. This demonstrates the potential of using pomsets for semantic analysis, and we are investigating how to use our framework for similar analysis.

Other related work includes the usage of event structures in the context of binary session types by Castellan and Yoshida [3] and multiparty by Castellani et al. [4]. Event structures and branching pomsets both feature a set of events with a causality relation and a choice mechanism. The main difference between the two approaches is in the choice mechanism. Event structures are based on a conflict relation on events, where two events in conflict cannot occur together in an execution and one of the two must be chosen. In contrast, we structure events in branching pomsets hierarchically. Given a branching pomset, one may construct an event structure by defining its conflict relation as all pairs of events that belong to different branches of some choice in the branching structure.

Acknowledgments

Funding by G. Cledou and J. Proença: European Regional Development Fund (ERDF), Operational Programme for Competitiveness and Internationalisation (COMPETE 2020), and Fundação para a Ciência e a Tecnologia (FCT): POCI-01-0145-FEDER-029946 (DaVinci). *Funding by S. Jongmans:* Netherlands Organisation of Scientific Research: 016.Veni.192.103. *Funding by J. Proença:* FCT, within the CISTER Research Unit: UIDB/04234/2020. FCT, I.P. (Portuguese Foundation for Science and Technology): PTDC/CCI-COM/4280/2021 (IBEX). ERDF and FCT, Portugal 2020 Partnership Agreement, Norte Portugal Regional Operational Programme (NORTE 2020): NORTE-01-0145-FEDER-028550 (REASSURE). ECSEL Joint Undertaking (JU): grant agreement No 876852 (VALU3S).

References

[1] Rajeev Alur, Kousha Etessami & Mihalis Yannakakis (2005): *Realizability and verification of MSC graphs*. *Theor. Comput. Sci.* 331(1), pp. 97–114.

[2] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In: *POPL*, ACM, pp. 263–274.

[3] Simon Castellan & Nobuko Yoshida (2019): *Two sides of the same coin: session types and game semantics: a synchronous side and an asynchronous side*. *Proc. ACM Program. Lang.* 3(POPL), pp. 27:1–27:29, doi:10.1145/3290340. Available at `https://doi.org/10.1145/3290340`.

[4] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2019): *Event Structure Semantics for Multiparty Sessions*. In Michele Boreale, Flavio Corradini, Michele Loreti & Rosario Pugliese, editors: *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, *Lecture Notes in Computer Science* 11665, Springer, pp. 340–363, doi:10.1007/978-3-030-21485-2_19. Available at `https://doi.org/10.1007/978-3-030-21485-2_19`.

[5] Luís Cruz-Filipe & Fabrizio Montesi (2020): *A core model for choreographic programming*. *Theor. Comput. Sci.* 802, pp. 38–66.

[6] Roberto Guanciale & Emilio Tuosto (2019): *Realisability of pomsets*. *J. Log. Algebraic Methods Program.* 108, pp. 69–89, doi:10.1016/j.jlamp.2019.06.003. Available at `https://doi.org/10.1016/j.jlamp.2019.06.003`.

[7] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, ACM, pp. 273–284.

[8] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67.

[9] ITU (2011): *ITU-T recommendation Z.120. Message Sequence Chart (MSC)*.

[10] Joost-Pieter Katoen & Lennard Lambert (1998): *Pomsets for MSC*. In Hartmut König & Peter Langendörfer, editors: *Formale Beschreibungstechniken für verteilte Systeme, 8. GI/ITG-Fachgespräch, Cottbus, 4. und 5. Juni 1998*, Verlag Shaker, pp. 197–207.

[11] Rumyana Neykova & Nobuko Yoshida (2017): *Let it recover: multiparty protocol-induced recovery*. In Peng Wu & Sebastian Hack, editors: *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*, ACM, pp. 98–108. Available at `http://dl.acm.org/citation.cfm?id=3033031`.

[12] Vaughan R. Pratt (1986): *Modeling concurrency with partial orders*. *Int. J. Parallel Program.* 15(1), pp. 33–71, doi:10.1007/BF01379149. Available at `https://doi.org/10.1007/BF01379149`.

[13] Arend Rensink & Heike Wehrheim (2001): *Process algebra with action dependencies*. *Acta Informatica* 38(3), pp. 155–234, doi:10.1007/s002360100070. Available at `https://doi.org/10.1007/s002360100070`.

[14] Davide Sangiorgi (2011): *Introduction to bisimulation and coinduction*. Cambridge University Press.

[15] Emilio Tuosto & Roberto Guanciale (2018): *Semantics of global view of choreographies*. *J. Log. Algebraic Methods Program.* 95, pp. 17–40, doi:10.1016/j.jlamp.2017.11.002. Available at `https://doi.org/10.1016/j.jlamp.2017.11.002`.

A Proofs from the paper

Lemma 1. *Let c_1 and c_2 be dependently guarded choreographies. Let $c_2 \xrightarrow{\ell} c_2'$ and $[\![c_2]\!] \xrightarrow{\checkmark_e} R_2'$ such that $\lambda(e) = \ell$ and $[\![c_2']\!] = R_2' - e$.*

(a) If $c_1 \xrightarrow{\checkmark_\ell} c_1'$ then $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_1']\!]\,;R_2'$.

(b) If $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} R_1'\,;R_2'$ then $c_1 \xrightarrow{\checkmark_{\lambda(e)}} c_1'$ and $[\![c_1']\!] = R_1'$.

Proof. This is a proof by induction on the structure of c_1. We assume both (a) and (b) to hold for all subexpressions of c_1.

(a)

- If $c_1 = \mathbf{0}$ then $[\![c_1]\!] = [\![c_1']\!] = \langle\emptyset,\emptyset,\emptyset,\emptyset\rangle$ and $[\![c_1\,;c_2]\!] = [\![c_2]\!]$ so the result holds trivially.
- If $c_1 = \mathtt{a}{\rightarrow}\mathtt{b}{:}\mathtt{x}$ then by Fig. 4 $subj(\ell) \notin \{\mathtt{a},\mathtt{b}\}$ and $c_1' = c_1$. By Fig. 9 the construction of $[\![c_1;c_2]\!]$ adds no dependencies between events in $[\![c_1]\!]$ and e, so $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_1]\!];R_2' = [\![c_1']\!];R_2'$.
- If $c_1 = \mathtt{ab?x}$ then we proceed analogously to the previous case.
- If $c_1 = c_3 \dagger c_4$ for $\dagger \in \{;,\|\}$ then by Fig. 4 $c_3 \xrightarrow{\checkmark_\ell} c_3'$ and $c_4 \xrightarrow{\checkmark_\ell} c_4'$ and $c_1' = c_3' \dagger c_4'$. By the induction hypothesis (a) $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_3']\!]\,;R_2'$ and $[\![c_4\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_4']\!]\,;R_2'$. By Fig. 7 $[\![c_2]\!] \sqsupseteq R_2'$, $[\![c_3]\!] \sqsupseteq [\![c_3']\!]$ and $[\![c_4]\!] \sqsupseteq [\![c_4']\!]$. By Lemma 8(i,iii) $[\![(c_3 \dagger c_4)\,;c_2]\!] \sqsupseteq ([\![c_3']\!] \dagger [\![c_4']\!])\,;R_2'$. Since $e \in$ a-min(R_2'), $e \in$ a-min$([\![c_3']\!]\,;R_2')$ and $e \in$ a-min$([\![c_4']\!]\,;R_2')$, it follows that $e \in$ a-min$(([\![c_3']\!] \dagger [\![c_4']\!])\,;R_2')$. Suppose there exists some R'' such that $([\![c_3]\!] \dagger [\![c_4]\!])\,;[\![c_2]\!] \sqsupseteq R'' \sqsupset ([\![c_3]\!] \dagger [\![c_4]\!])\,;R_2'$. If $e \in$ a-min(R'') then it follows from Lemma 8(iv) that either $[\![c_3]\!]\,;[\![c_2]\!] \sqsupseteq R_3' \sqsupset [\![c_3']\!]\,;R_2'$ and $e \in$ a-min(R_3') or analogously for c_4. This contradicts our observation that $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_3']\!];R_2'$, or analogously for c_4. We conclude that $e \notin$ a-min(R'') and then by Fig. 7 $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_1']\!]\,;R_2'$.
- If $c_1 = c_3 + c_4$, we can distinguish three cases:
  - If $c_3 \xrightarrow{\checkmark_\ell} c_3'$ and $c_4 \xrightarrow{\checkmark_\ell} c_4'$ then $c_1' = c_3' + c_4'$. We then proceed analogously to the previous case, applying Lemma 8(ii,v) instead of Lemma 8(i,iii,iv).
  - If $c_3 \xrightarrow{\checkmark_\ell} c_3'$ but $c_4 \not\xrightarrow{\checkmark_\ell}$ then $c_1' = c_3'$. By the induction hypothesis (a) $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_3']\!];R_2'$, from which it follows that $e \in$ a-min$([\![c_3']\!]\,;R_2')$. By the induction hypothesis (b) $[\![c_4\,;c_2]\!] \not\xrightarrow{\checkmark_e}$ since it would otherwise contradict the premise that $c_4 \not\xrightarrow{\checkmark_\ell}$. By Lemma 8(ii,iii) $[\![c_1\,;c_2]\!] \sqsupseteq [\![c_3']\!];R_2'$. Suppose that there exists some R'' such that $[\![c_1\,;c_2]\!] \sqsupseteq R'' \sqsupset [\![c_3']\!];R_2'$. By Lemma 8(iv) $R'' = R_1''\,;R_2''$ for some $[\![c_1]\!] \sqsupseteq R_1'' \sqsupseteq [\![c_3']\!]$ and $[\![c_2]\!] \sqsupseteq R_2'' \sqsupseteq R_2'$. If $R_2'' \neq R_2'$ then $e \notin$ a-min(R_2'') and $e \notin$ a-min(R''). By Lemma 8(v) either $R_1'' = R_4''$ for some $[\![c_4]\!] \sqsupseteq R_4'' \sqsupseteq [\![c_3']\!]$, which is clearly impossible, or $R_1'' = R_3''$ for some $[\![c_3]\!] \sqsupseteq R_3'' \sqsupseteq [\![c_3']\!]$, in which case $e \notin$ a-min(R'') since this would otherwise contradict $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_3']\!]\,;R_2'$, or $R_1'' = R_3'' + R_4''$, in which case either $e \notin$ a-min(R'') or $e \in$ a-min$(R_4'';R_2'')$, which contradicts $[\![c_4\,;c_2]\!] \not\xrightarrow{\checkmark_e}$. Then by Fig. 7 $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_3']\!]\,;R_2' = [\![c_1']\!]\,;R_2'$.
  - If $c_3 \not\xrightarrow{\checkmark_\ell}$ and $c_4 \xrightarrow{\checkmark_\ell} c_4'$ then we proceed analogously to the previous case.
- If $c_1 = c_3^*$ for some c_3 then by Fig. 4 we can distinguish two cases:
  - If $c_3 \xrightarrow{\checkmark_\ell} c_3$ then $c_1' = c_1$. Since c_1 is dependently guarded, it follows that the subject of ℓ does not occur in c_3 or in c_1. Then by Fig. 9 there are no dependencies between any event in $[\![c_1]\!]$ and e in $[\![c_1\,;c_2]\!]$. It follows that $[\![c_1\,;c_2]\!] \sqsupseteq [\![c_1]\!]\,;R_2'$ and $e \in$ a-min$([\![c_1]\!]\,;R_2')$. Since $[\![c_2]\!] \xrightarrow{\checkmark_e} R_2'$ there exists no R_2'' such that $[\![c_2]\!] \sqsupseteq R_2'' \sqsupset R_2'$ and $e \in$ a-min(R_2''). It then follows from Fig. 7 that $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_1']\!]\,;R_2'$.
  - If $c_3 \not\xrightarrow{\checkmark_\ell} c_3$ then $c_1' = \mathbf{0}$. By Fig. 9 $[\![c_1]\!] = [\![(c_3\,;c_3^*) + \mathbf{0}]\!]$. By Lemma 8(ii) $[\![c_1]\!] \sqsupseteq [\![\mathbf{0}]\!]$ and then by Lemma 8(iii) $[\![c_1]\!]\,;[\![c_2]\!] \sqsupseteq [\![\mathbf{0}]\!]\,;R_2' = R_2'$. Since $[\![c_2]\!] \xrightarrow{\checkmark_e} R_2'$, by Fig. 7 $e \in$ a-min(R_2').

Suppose there exists some $R'_1 \neq [\![\mathbf{0}]\!]$ such that $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} R'_1\,;R'_2$. Then by the induction hypothesis (b) $c_1 \xrightarrow{\checkmark_{\lambda(e)}} c'_1$ and $[\![c'_1]\!] = R'_1$ for some c'_1. Since $R'_1 \neq [\![\mathbf{0}]\!]$, $c'_1 \neq \mathbf{0}$, which contradicts our earlier statement that $c_1 = \mathbf{0}$. We conclude that there exists no such R'_1 and that by Fig. 7 $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c'_1]\!]\,;R'_2$.

(b) By Lemma 8(iv) $[\![c_1]\!] \sqsupseteq R'_1$.

- If $c_1 = \mathbf{0}$ then $[\![c_1]\!] = [\![c'_1]\!] = R'_1 = \langle \emptyset,\emptyset,\emptyset,\emptyset\rangle$. By Fig. 4 $\mathbf{0} \xrightarrow{\checkmark_{\lambda(e)}} \mathbf{0}$ so the result holds trivially.
- If $c_1 = \mathtt{a}{\rightarrow}\mathtt{b}{:}\mathtt{x}$ then by Fig. 7 $R'_1 = [\![c_1]\!]$ since there is no rule to refine it and $subj(\lambda(e)) \notin \{\mathtt{a},\mathtt{b}\}$ since $e \in$ a-min$(R'_1\,;R'_2)$. Then by Fig. 4 $c_1 \xrightarrow{\checkmark_{\lambda(e)}} c_1 = c'_1$.
- If $c_1 = \mathtt{ab}?\mathtt{x}$ then we proceed analogously to the previous case.
- If $c_1 = c_3 \dagger c_4$ for $\dagger \in \{;,\|\}$ then by Lemma 8(iv) $R'_1 = R'_3 \dagger R'_4$ for some $[\![c_3]\!] \sqsupseteq R'_3$ and $[\![c_4]\!] \sqsupseteq R'_4$. Suppose there exists some R''_3 such that $[\![c_3\,;c_2]\!] \sqsupseteq R''_3\,;R'_2 \sqsupseteq R'_3\,;R'_2$ and $e \in$ a-min$(R''_3\,;R'_2)$. It would follow from Lemma 8(i,iii) that $[\![c_1\,;c_2]\!] \sqsupseteq (R''_3 \dagger R'_4)\,;R'_2 \sqsupset R'_1\,;R'_2$ and $e \in$ a-min$((R''_3 \dagger R'_4)\,;R'_2)$, which contradicts our premise that $[\![c_1\,;c_2]\!] \xrightarrow{\checkmark_e} R'_1\,;R'_2$. It thus follows from Fig. 7 that $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} R'_3\,;R'_2$ and similarly for c_4. By the induction hypothesis (b) $c_3 \xrightarrow{\checkmark_{\lambda(e)}} c'_3$ such that $[\![c'_3]\!] = R'_3$ and similarly for c_4. Then by Fig. 4 $c_3 \dagger c_4 \xrightarrow{\checkmark_{\lambda(e)}} c'_3 \dagger c'_4$.
- If $c_1 = c_3 + c_4$ then by Lemma 8(v) we can distinguish three cases:
  - If $R'_1 = R'_3 + R'_4$ for some $R'_3 \sqsupseteq [\![c_3]\!]$ and $R'_4 \sqsupseteq [\![c_4]\!]$ then by Lemma 8(iii) $[\![c_3\,;c_2]\!] \sqsupseteq R'_3\,;R'_2$ and similarly for c_4. Analogously to the previous case it follows that $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} R'_3\,;R'_2$ and by the induction hypothesis (b) that $c_3 \xrightarrow{\checkmark_{\lambda(e)}} c'_3$ and $[\![c'_3]\!] = R'_3$, and similarly for c_4. Then by Fig. 4 $c_3 + c_4 \xrightarrow{\checkmark_{\lambda(e)}} c'_3 + c'_4$ and by Fig. 9 $[\![c'_3 + c'_4]\!] = R'_3 + R'_4$.
  - If $R'_1 \sqsupseteq [\![c_3]\!]$ then analogously to the previous case it follows that $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} R'_1\,;R'_2$. By the induction hypothesis (b) $c_3 \xrightarrow{\checkmark_{\lambda(e)}} c'_3$ and $R'_1 = [\![c'_3]\!]$.
    Suppose that $c_4 \xrightarrow{\checkmark_{\lambda(e)}} c'_4$ for some c'_4. Then by the induction hypothesis (a) also $[\![c_4\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c'_4]\!]\,;R'_2$. It would follow from Fig. 7 that $[\![(c_3+c_4)\,;c_2]\!] \xrightarrow{\checkmark_e} ([\![c'_3]\!] + [\![c'_4]\!])\,;R'_2$. However, since $([\![c'_3]\!] + [\![c'_4]\!])\,;R'_2 \sqsupseteq [\![c'_3]\!]\,;R'_2$ this contradicts our premise that $R'_1 = [\![c'_3]\!]$. We conclude that $c_4 \not\xrightarrow{\checkmark_{\lambda(e)}}$ and then by Fig. 4 $c_3 + c_4 \xrightarrow{\checkmark_{\lambda(e)}} c'_3$.
  - If $R'_1 \sqsupseteq [\![c_4]\!]$ then we proceed analogously to the previous case.
- If $c_1 = c_3^*$ then recall that by Fig. 9 $[\![c_3^*]\!] = [\![(c_3\,;c_3^*) + \mathbf{0}]\!]$. We can distinguish two cases:
  - If $R'_1 = [\![c_1]\!]$ then analogously to the previous cases it follows that $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} [\![c_3]\!]\,;R'_2$ and by the induction hypothesis (b) $c_3 \xrightarrow{\checkmark_{\lambda(e)}} c_3$. Then by Fig. 4 $c_1 \xrightarrow{\checkmark_{\lambda(e)}} c_1$.
  - If $R'_1 \neq [\![c_1]\!]$ then suppose that $[\![c_3\,;c_2]\!] \xrightarrow{\checkmark_e} R'_3\,;R'_2$ for some $[\![c_3]\!] \sqsupset R'_3$. It would follow from the induction hypothesis (b) that $c_3 \xrightarrow{\checkmark_{\lambda(e)}} c'_3$ such that $[\![c'_3]\!] = R'_3$. Then $c'_3 \neq c_3$, which is contradictory since $c_1 = c_3^*$ is dependently guarded. It thus follows that $[\![c_3\,;c_2]\!] \not\xrightarrow{\checkmark_e}$ and that $R'_1 = [\![\mathbf{0}]\!]$. By the induction hypothesis (a) $c_3 \not\xrightarrow{\checkmark_{\lambda(e)}}$ and then by Fig. 4 $c_1 \xrightarrow{\checkmark_{\lambda(e)}} \mathbf{0}$. □

Lemma 2. *Let c be a dependently guarded choreography. If $c \xrightarrow{\ell} c'$ then $[\![c]\!] \xrightarrow{\ell} [\![c']\!]$.*

Proof. This is a proof by structural induction on c.

- Suppose $c \in \{\mathbf{0}, \mathtt{a}{\rightarrow}\mathtt{b}{:}\mathtt{x}, \mathtt{ab}?\mathtt{x}\}$. Then the result holds trivially.
- Suppose $c = c_1 \parallel c_2$. If $c \xrightarrow{\ell} c'$, then without loss of generality $c_1 \xrightarrow{\ell} c'_1$ and $c' = c'_1 \parallel c_2$ (the other case is analogous). By the induction hypothesis $[\![c_1]\!] \xrightarrow{\ell} [\![c'_1]\!]$, so by Fig. 7 $[\![c_1]\!] \xrightarrow{\checkmark_e} R'$ such that $[\![c'_1]\!] = R' - e$ and $\lambda(e) = \ell$. It follows from Lemma 9(i) that $[\![c'_1]\!] \parallel [\![c_2]\!] \xrightarrow{\checkmark_e} R' \parallel [\![c_2]\!]$ and then by Fig. 7 $[\![c]\!] = [\![c_1]\!] \parallel [\![c_2]\!] \xrightarrow{\ell} (R' - e) \parallel [\![c_2]\!] = [\![c'_1]\!] \parallel [\![c_2]\!] = [\![c']\!]$.

- Suppose $c = c_1 + c_2$. If $c \xrightarrow{\ell} c'$, then without loss of generality $c_1 \xrightarrow{\ell} c'$ (the other case if analogous). By the induction hypothesis $[\![c_1]\!] \xrightarrow{\ell} [\![c']\!]$, so by Fig. 7 $[\![c_1]\!] \xrightarrow{\checkmark_e} R'$ such that $[\![c']\!] = R' - e$ and $\lambda(e) = \ell$. It follows from Lemma 9(ii) that $[\![c_1]\!] + [\![c_2]\!] \xrightarrow{\checkmark_e} R'$ and then by Fig. 7 $[\![c]\!] = [\![c_1]\!] + [\![c_2]\!] \xrightarrow{\ell} R' - e = [\![c']\!]$.
- Suppose $c = c_1^*$. If $c \xrightarrow{\ell} c'$, then $c_1 \xrightarrow{\ell} c_1'$ and $c' = c_1' \,;c_1^*$. By the induction hypothesis $[\![c_1]\!] \xrightarrow{\ell} [\![c_1']\!]$, so by Fig. 7 $[\![c_1]\!] \xrightarrow{\checkmark_e} R'$ such that $[\![c_1']\!] = R' - e$ and $\lambda(e) = \ell$. Since $[\![c_1^*]\!] = [\![(c_1 \,; c_1^*) + \mathbf{0}]\!]$, it follows from Lemma 9(ii–iii) that $[\![(c_1 \,; c_1^*) + \mathbf{0}]\!] \xrightarrow{\checkmark_e} R' \,; [\![c_1^*]\!]$ and then by Fig. 7 $[\![c]\!] = [\![(c_1 \,; c_1^*) + \mathbf{0}]\!] \xrightarrow{\ell} (R' - e) \,; [\![c_1^*]\!] = [\![c_1']\!] \,; [\![c_1^*]\!] = [\![c']\!]$.
- Finally, suppose $c = c_1 \,; c_2$. If $c \xrightarrow{\ell} c'$, we can distinguish two cases:
  - Suppose $c_1 \xrightarrow{\ell} c_1'$ and $c' = c_1';c_2$. By the induction hypothesis $[\![c_1]\!] \xrightarrow{\ell} [\![c_1']\!]$, so by Fig. 7 $[\![c_1]\!] \xrightarrow{\checkmark_e} R'$ such that $[\![c_1']\!] = R' - e$ and $\lambda(e) = \ell$. It follows from Lemma 9(iii) that $[\![c_1]\!] \,; [\![c_2]\!] \xrightarrow{\checkmark_e} R';[\![c_2]\!]$ and then by Fig. 7 $[\![c]\!] = [\![c_1]\!] \,; [\![c_2]\!] \xrightarrow{\ell} (R' - e) \,; [\![c_2]\!] = [\![c_1']\!] \,; [\![c_2]\!] = [\![c']\!]$.
  - Suppose $c_1 \xrightarrow{\checkmark_\ell} c_1'$, $c_2 \xrightarrow{\ell} c_2'$ and $c' = c_1' \,; c_2'$. By the induction hypothesis $[\![c_2]\!] \xrightarrow{\ell} [\![c_2']\!]$ and then it follows from Lemma 1(a) that $[\![c]\!] = [\![c_1]\!] \,; [\![c_2]\!] \xrightarrow{\checkmark_e} [\![c_1']\!] \,; R_2' \xrightarrow{\ell} [\![c_1']\!] \,; [\![c_2']\!] = [\![c']\!]$. □

Lemma 3. *Let c be a dependently guarded choreography. If $[\![c]\!] \xrightarrow{\ell} R'$ for some R' then $c \xrightarrow{\ell} c'$ such that $R' = [\![c']\!]$.*

Proof. This is a proof by structural induction on c. Let $R = [\![c]\!]$.

- Suppose $c \in \{\mathbf{0}, \mathtt{a{\rightarrow}b{:}x}, \mathtt{ab?x}\}$. Then the result holds trivially.
- Suppose $c = c_1 \parallel c_2$. If $R \xrightarrow{\ell} R'$, then without loss of generality $[\![c_1]\!] \xrightarrow{\ell} R_1'$ and $R' = R_1' \parallel [\![c_2]\!]$ (the other case is analogous). By the induction hypothesis there exists some c_1' such that $c_1 \xrightarrow{\ell} c_1'$ such that $R_1' = [\![c_1']\!]$. Then by Fig. 4 $c_1 \parallel c_2 \xrightarrow{\ell} c_1' \parallel c_2 = c'$, and $[\![c']\!] = R'$.
- Suppose $c = c_1 + c_2$. If $R \xrightarrow{\ell} R'$, then without loss of generality $[\![c_1]\!] \xrightarrow{\ell} R'$ (the other case is analogous). By the induction hypothesis there then exists some c' such that $c_1 \xrightarrow{\ell} c'$ and $[\![c']\!] = R'$ and then by Fig. 4 $c_1 + c_2 \xrightarrow{\ell} c'$.
- Suppose $c = c_1^*$. If $R \xrightarrow{\ell} R'$, then it follows from Lemma 10 that $[\![c_1]\!] \xrightarrow{\ell} R_1'$ and $R' = R_1' \,; [\![c_1^*]\!]$. By the induction hypothesis there exists some c_1' such that $c_1 \xrightarrow{\ell} c_1'$ and $R_1' = [\![c_1']\!]$. Then by Fig. 4 $c_1^* \xrightarrow{\ell} c_1' \,; c_1^* = c'$, and $R' = [\![c']\!]$.
- Finally, suppose $c = c_1 \,; c_2$. If $R \xrightarrow{\ell} R'$, then by Fig. 7 $R \xrightarrow{\checkmark_e} R''$ such that $R' = R'' - e$ and $\lambda(e) = \ell$. By Lemma 8(iv) $R'' = R_1' \,; R_2'$ for some $[\![c_1]\!] \sqsupseteq R_1'$ and $[\![c_2]\!] \sqsupseteq R_2'$. We can distinguish two cases:
  - Suppose e is an event in $[\![c_1]\!]$. Suppose $[\![c_1]\!] \sqsupseteq R_1'' \sqsupset R_1'$ for some R_1''. Then $e \notin$ a-min(R_1''). If it were, then also $[\![c_1]\!] \,; [\![c_2]\!] \sqsupseteq R_1'' \,; R_2' \sqsupset R_1' \,; R_2'$ and $e \in$ a-min$(R_1'' \,; R_2')$, which contradicts Fig. 7. It follows from Fig. 7 that $[\![c_1]\!] \xrightarrow{\checkmark_e} R_1'$ and $[\![c_1]\!] \xrightarrow{\ell} R_1' - e$. By the induction hypothesis there exists some c_1' such that $c_1 \xrightarrow{\ell} c_1'$ and $[\![c_1']\!] = R_1' - e$. By Fig. 4 $c_1 \,; c_2 \xrightarrow{\ell} c_1' \,; c_2 = c'$ and then $R' = [\![c']\!]$.
  - Suppose e is an event in $[\![c_2]\!]$. Suppose $[\![c_2]\!] \sqsupseteq R_2'' \sqsupset R_2'$ for some R_2''. Then $e \notin$ a-min(R_2''). If it were, then also $[\![c_1]\!] \,; [\![c_2]\!] \sqsupseteq R_1' \,; R_2'' \sqsupset R_1' \,; R_2'$ and $e \in$ a-min$(R_1' \,; R_2'')$, which contradicts Fig. 7. It follows from Fig. 7 that $[\![c_2]\!] \xrightarrow{\checkmark_e} R_2'$ and $[\![c_2]\!] \xrightarrow{\ell} R_2' - e$. By the induction hypothesis there exists some c_2' such that $c_2 \xrightarrow{\ell} c_2'$ and $[\![c_2']\!] = R_2' - e$. It then follows from Lemma 1(b) that $c_1 \xrightarrow{\checkmark_\ell} c_1'$ and $[\![c_1']\!] = R_1'$. Then by Fig. 4, $c_1 \,; c_2 \xrightarrow{\ell} c_1' \,; c_2' = c'$ and $[\![c']\!] = R'$. □

Lemma 4. *Let c be a dependently guarded choreography. If $c{\downarrow}$ then $[\![c]\!]{\downarrow}$.*

Proof. This is a proof by structural induction on c.

- If $c = \mathbf{0}$ then both c and $[\![c]\!]$ can terminate.
- If $c = \mathtt{a}{\to}\mathtt{b}{:}\mathtt{x}$ or $c = \mathtt{ab}?\mathtt{x}$ then neither c or $[\![c]\!]$ can terminate.
- If $c = c_1 \dagger c_2$ for $\dagger \in \{;, \|\}$ then by Fig. 4 $c_1{\downarrow}$ and $c_2{\downarrow}$. By the induction hypothesis $[\![c_1]\!]{\downarrow}$ and $[\![c_2]\!]{\downarrow}$. By Fig. 7 $[\![c_1]\!].\mathcal{B} \sqsupseteq \emptyset$ and $[\![c_2]\!].\mathcal{B} \sqsupseteq \emptyset$. By Fig. 9 $[\![c_1 \dagger c_2]\!].\mathcal{B} = [\![c_1]\!].\mathcal{B} \cup [\![c_2]\!].\mathcal{B}$ and by Fig. 7 $[\![c_1 \dagger c_2]\!].\mathcal{B} \sqsupseteq \emptyset$ and $[\![c_1 \dagger c_2]\!]{\downarrow}$.
- If $c = c_1 + c_2$ then by Fig. 4 either $c_1{\downarrow}$ or $c_2{\downarrow}$. Without loss of generality we assume $c_1{\downarrow}$; the other case is analogous. By the induction hypothesis $[\![c_1]\!]{\downarrow}$ and by Fig. 7 $[\![c_1]\!].\mathcal{B} \sqsupseteq \emptyset$. By Fig. 7 $[\![c_1 + c_2]\!].\mathcal{B} \sqsupseteq \emptyset$ and then $[\![c_1 + c_2]\!]{\downarrow}$.
- If $c = c_1^*$ then $c{\downarrow}$ by Fig. 4. By Fig. 9 $[\![c]\!] = [\![(c_1\,; c_1^*) + \mathbf{0}]\!]$. Since $[\![\mathbf{0}]\!].\mathcal{B} = \emptyset$, it follows from Fig. 7 that $[\![c]\!].\mathcal{B} \sqsupseteq \emptyset$ and then $[\![c]\!]{\downarrow}$. □

Lemma 5. *Let c be a dependently guarded choreography. If $[\![c]\!]{\downarrow}$ then $c{\downarrow}$.*

Proof. This is a proof by structural induction on c.

- If $c = \mathbf{0}$ then both c and $[\![c]\!]$ can terminate.
- If $c = \mathtt{a}{\to}\mathtt{b}{:}\mathtt{x}$ or $c = \mathtt{ab}?\mathtt{x}$ then neither c or $[\![c]\!]$ can terminate.
- If $c = c_1 \dagger c_2$ for $\dagger \in \{;, \|\}$ and $[\![c]\!]{\downarrow}$ then by Fig. 7 $[\![c_1 \dagger c_2]\!].\mathcal{B} \sqsupseteq \emptyset$. It follows from Lemma 7(ii) that $\emptyset = \mathcal{B}_1' \cup \mathcal{B}_2'$ such that $[\![c_1]\!].\mathcal{B} \sqsupseteq \mathcal{B}_1'$ and $[\![c_2]\!].\mathcal{B} \sqsupseteq \mathcal{B}_2'$. It follows that $[\![c_1]\!].\mathcal{B} \sqsupseteq \emptyset$ and $[\![c_2]\!].\mathcal{B} \sqsupseteq \emptyset$, so by Fig. 7 $[\![c_1]\!]{\downarrow}$ and $[\![c_2]\!]{\downarrow}$. By the induction hypothesis $c_1{\downarrow}$ and $c_2{\downarrow}$ and then by Fig. 4 $c_1 \dagger c_2{\downarrow}$.
- If $c = c_1 + c_2$ and $c{\downarrow}$ then by Fig. 7 $[\![c_1 + c_2]\!].\mathcal{B} \sqsupseteq \emptyset$. By Fig. 9 $[\![c_1 + c_2]\!].\mathcal{B} = \{\{[\![c_1]\!].\mathcal{B}, [\![c_2]\!].\mathcal{B}\}\}$. By Lemma 7(iii) either:
  - $\emptyset = \{\{\mathcal{B}_1', \mathcal{B}_2'\}\}$ for some $[\![c_1]\!].\mathcal{B} \sqsupseteq \mathcal{B}_1'$ and $[\![c_2]\!].\mathcal{B} \sqsupseteq \mathcal{B}_2'$, which is a clear contradiction; or
  - $[\![c_1]\!].\mathcal{B} \sqsupseteq \emptyset$, in which case $[\![c_1]\!]{\downarrow}$ and by the induction hypothesis $c_1{\downarrow}$ and then by Fig. 4 $c_1 + c_2{\downarrow}$; or
  - $[\![c_2]\!].\mathcal{B} \sqsupseteq \emptyset$, which is analogous to the previous case.
- If $c = c_1^*$ then, as in Lemma 4, both $[\![c]\!]{\downarrow}$ and $c{\downarrow}$. □

B Additional proofs

Lemma 7. *Let $\mathcal{B}_1, \mathcal{B}_2$ be branching structures.*

(i) If $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_1 \uplus \mathcal{B}_2$ is defined, then $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\ddagger \cup \mathcal{B}_2$.

(ii) If $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}^\ddagger$, then $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger$ and $\mathcal{B}_1^\ddagger \cup \mathcal{B}_2^\ddagger = \mathcal{B}^\ddagger$, for some $\mathcal{B}_1^\ddagger, \mathcal{B}_2^\ddagger$.

(iii) If $\{\{\mathcal{B}_1, \mathcal{B}_2\}\} \sqsupseteq \mathcal{B}^\ddagger$, then either $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger$ and $\{\{\mathcal{B}_1^\ddagger, \mathcal{B}_2^\ddagger\}\} = \mathcal{B}^\ddagger$, for some $\mathcal{B}_1^\ddagger, \mathcal{B}_2^\ddagger$, or $\mathcal{B}_1 \sqsupseteq \mathcal{B}^\ddagger$, or $\mathcal{B}_2 \sqsupseteq \mathcal{B}^\ddagger$.

Proof.

(i) Recall $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger$. Then, by the definition of refinement:

- **Base:** Refl, such that $\mathcal{B}_1 = \mathcal{B}_1^\ddagger$.
  Recall $\mathcal{B}_1 \uplus \mathcal{B}_2$ is defined. Then, by Refl, $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1 \cup \mathcal{B}_2$. Then, $\boxed{\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\ddagger \cup \mathcal{B}_2}$.

- **Base:** CHOICE, such that $\mathcal{B}_1 = \{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}$ and $\mathcal{B}_1^\ddagger = \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}$ and $1 \leq i \leq m$, for some $\hat{\mathcal{B}}, \hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m, i, m$.
  - Recall $\mathcal{B}_1 \uplus \mathcal{B}_2$ is defined. Then, $(\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}) \uplus \mathcal{B}_2$ is defined. Then, $\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus (\hat{\mathcal{B}} \uplus \mathcal{B}_2)$ is defined.
  - Recall $\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus (\hat{\mathcal{B}} \uplus \mathcal{B}_2)$ is defined, and $1 \leq i \leq m$. Then, by CHOICE, $\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus (\hat{\mathcal{B}} \uplus \mathcal{B}_2) \sqsupseteq \hat{\mathcal{B}}_i \cup (\hat{\mathcal{B}} \uplus \mathcal{B}_2)$. Then, $\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \cup (\hat{\mathcal{B}} \cup \mathcal{B}_2) \sqsupseteq \hat{\mathcal{B}}_i \cup (\hat{\mathcal{B}} \cup \mathcal{B}_2)$. Then, $(\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \cup \hat{\mathcal{B}}) \cup \mathcal{B}_2 \sqsupseteq (\hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}) \cup \mathcal{B}_2$. Then, $\boxed{\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\ddagger \cup \mathcal{B}_2}$.
- **Step:** TRANS, such that $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}_1^\ddagger$, for some $\mathcal{B}_1^\dagger$.
  - Recall $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger$ and $\mathcal{B}_1 \uplus \mathcal{B}_2$ is defined. Then, by induction, $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\dagger \cup \mathcal{B}_2$.
  - Recall $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\dagger \cup \mathcal{B}_2$. Then, $\mathcal{B}_1^\dagger \cup \mathcal{B}_2$ is defined.
  - Recall $\mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_1^\dagger \cup \mathcal{B}_2$ is defined. Then, by induction, $\mathcal{B}_1^\dagger \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\ddagger \cup \mathcal{B}_2$.
  - Recall $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\dagger \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\ddagger \cup \mathcal{B}_2$. Then, by TRANS, $\boxed{\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1^\ddagger \cup \mathcal{B}_2}$.
- **Step:** CONGR. Similar to case CHOICE. □

(ii) Recall $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}^\ddagger$. Then, by the definition of refinement:

- **Base:** REFL, such that $\mathcal{B}_1 \cup \mathcal{B}_2 = \mathcal{B}^\ddagger$.
  - By REFL, $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1$. Then, $\mathcal{B}_1^\ddagger = \mathcal{B}_1$ and $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_1^\ddagger}$.
  - By REFL, $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2$. Then, $\mathcal{B}_2^\ddagger = \mathcal{B}_2$ and $\boxed{\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_2^\ddagger}$.
  - Recall $\mathcal{B}_1 \cup \mathcal{B}_2 = \mathcal{B}^\ddagger$. Then, $\boxed{\mathcal{B}_1^\ddagger \cup \mathcal{B}_2^\ddagger = \mathcal{B}^\ddagger}$.
- **Base:** CHOICE, such that $\mathcal{B}_1 \cup \mathcal{B}_2 = \{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}$ and $\mathcal{B}^\ddagger = \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}$ and $1 \leq i \leq m$, for some $\hat{\mathcal{B}}, \hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m, i, m$.
  Recall $\mathcal{B}_1 \cup \mathcal{B}_2 = \{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}$. Then:
  - **Case 1:** $\mathcal{B}_1 = \{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}'$ and $\mathcal{B}_2 = \hat{\mathcal{B}}''$ and $\hat{\mathcal{B}} = \hat{\mathcal{B}}' \cup \hat{\mathcal{B}}''$, for some $\hat{\mathcal{B}}', \hat{\mathcal{B}}''$.
    * Recall $1 \leq i \leq m$. Then, by CHOICE, $\{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}' \sqsupseteq \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}'$. Then, $\mathcal{B}_1 \sqsupseteq \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}'$. Then, $\mathcal{B}_1^\ddagger = \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}'$ and $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_1^\ddagger}$.
    * By REFL, $\hat{\mathcal{B}}'' \sqsupseteq \hat{\mathcal{B}}''$. Then, $\mathcal{B}_2 \sqsupseteq \hat{\mathcal{B}}''$. Then, $\mathcal{B}_2^\ddagger = \hat{\mathcal{B}}''$ and $\boxed{\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_2^\ddagger}$.
    * Recall $\mathcal{B}^\ddagger = \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}$. Then, $\mathcal{B}^\ddagger = \hat{\mathcal{B}}_i \cup \hat{\mathcal{B}}' \cup \hat{\mathcal{B}}''$. Then, $\boxed{\mathcal{B}_1^\ddagger \cup \mathcal{B}_2^\ddagger = \mathcal{B}^\ddagger}$.
  - **Case 2:** $\mathcal{B}_1 = \hat{\mathcal{B}}'$ and $\mathcal{B}_2 = \{\{\hat{\mathcal{B}}_1, \dots, \hat{\mathcal{B}}_m\}\} \uplus \hat{\mathcal{B}}''$ and $\hat{\mathcal{B}} = \hat{\mathcal{B}}' \cup \hat{\mathcal{B}}''$, for some $\hat{\mathcal{B}}', \hat{\mathcal{B}}''$. Similar to case 1.
- **Step:** TRANS, such that $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}^\dagger \sqsupseteq \mathcal{B}^\ddagger$, for some $\mathcal{B}^\dagger$.
  - Recall $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}^\dagger$. Then, by induction, $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger$ and $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\dagger$ and $\mathcal{B}^\dagger = \mathcal{B}_1^\dagger \cup \mathcal{B}_2^\dagger$, for some $\mathcal{B}_1^\dagger, \mathcal{B}_2^\dagger$.
  - Recall $\mathcal{B}^\dagger \sqsupseteq \mathcal{B}^\ddagger$. Then, $\mathcal{B}_1^\dagger \cup \mathcal{B}_2^\dagger \sqsupseteq N^\ddagger$. Then, by induction, $\mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_2^\dagger \sqsupseteq \mathcal{B}_2^\ddagger$ and $\boxed{\mathcal{B}^\ddagger = \mathcal{B}_1^\ddagger \cup \mathcal{B}_2^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_1^\ddagger, \mathcal{B}_2^\ddagger}$.
  - Recall $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}_1^\ddagger$. Then, by TRANS, $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger}$.
  - Recall $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\dagger \sqsupseteq \mathcal{B}_2^\ddagger$. Then, by TRANS, $\boxed{\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger}$. □
- **Step:** CONGR. Similar to case CHOICE.

(iii) Recall $\{\{\mathcal{B}_1, \mathcal{B}_2\}\} \sqsupseteq \mathcal{B}^\ddagger$. Then, by the definition of refinement:

- **Base:** REFL, such that $\{\{\mathcal{B}_1, \mathcal{B}_2\}\} = \mathcal{B}^\ddagger$.
  - By REFL, $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1$. Then, $\mathcal{B}_1^\ddagger = \mathcal{B}_1$ and $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_1^\ddagger}$.
  - By REFL, $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2$. Then, $\mathcal{B}_2^\ddagger = \mathcal{B}_2$ and $\boxed{\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_2^\ddagger}$.

  - Recall $\{\{\mathcal{B}_1,\mathcal{B}_2\}\} = \mathcal{B}^\ddagger$. Then, $\boxed{\{\{\mathcal{B}_1^\ddagger,\mathcal{B}_2^\ddagger\}\} = \mathcal{B}^\ddagger}$.
- **Base:** CHOICE, such that $\mathcal{B}^\ddagger = \mathcal{B}_i$ and $1 \leq i \leq 2$.
  Recall $1 \leq i \leq 2$. Then:
  - **Case:** $i = 1$.
    By REFL, $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1$. Then, $\mathcal{B}_1 \sqsupseteq \mathcal{B}_i$. Then, $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}^\ddagger}$.
  - **Case:** $i = 2$. Similar to case $i = 1$.
- **Step:** TRANS, such that $\{\{\mathcal{B}_1,\mathcal{B}_2\}\} \sqsupseteq \mathcal{B}^\dagger \sqsupseteq \mathcal{B}^\ddagger$, for some $\mathcal{B}^\dagger$.
  - Recall $\{\{\mathcal{B}_1,\mathcal{B}_2\}\} \sqsupseteq \mathcal{B}^\dagger$. Then, by induction:
    * **Case 1:** $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger$ and $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\dagger$ and $\{\mathcal{B}_1^\dagger,\mathcal{B}_2^\dagger\} = \mathcal{B}$, for some $\mathcal{B}_1^\dagger,\mathcal{B}_2^\dagger$.
      Recall $\mathcal{B}^\dagger \sqsupseteq \mathcal{B}^\ddagger$. Then, $\{\mathcal{B}_1^\dagger,\mathcal{B}_2^\dagger\} \sqsupseteq \mathcal{B}^\ddagger$. Then, by induction:
      · **Case 1a:** $\mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_2^\dagger \sqsupseteq \mathcal{B}_2^\ddagger$ and $\boxed{\{\mathcal{B}_1^\ddagger,\mathcal{B}_2^\ddagger\} = \mathcal{B}^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_1^\ddagger,\mathcal{B}_2^\ddagger}$.
      Recall $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\dagger \sqsupseteq \mathcal{B}_2^\ddagger$. Then, by TRANS, $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger$ and $\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger$.
      · **Case 1b:** $\mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}^\ddagger$.
      Recall $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\dagger \sqsupseteq \mathcal{B}^\ddagger$. Then, by TRANS, $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}^\ddagger}$.
      · **Case 1c:** $\mathcal{B}_2^\dagger \sqsupseteq \mathcal{B}^\ddagger$. Similar to case 1b.
    * **Case 2:** $\mathcal{B}_1 \sqsupseteq \mathcal{B}^\dagger$.
      Recall $\mathcal{B}_1 \sqsupseteq \mathcal{B}^\dagger \sqsupseteq \mathcal{B}^\ddagger$. Then, by TRANS, $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}^\ddagger}$.
    * **Case 3:** $\mathcal{B}_2 \sqsupseteq \mathcal{B}^\dagger$. Similar to case 2.
- **Step:** CONGR, such that $\boxed{\mathcal{B}^\ddagger = \{\{\mathcal{B}_1^\ddagger,\mathcal{B}_2^\ddagger\}\}}$ and $\boxed{\mathcal{B}_1 \sqsupseteq \mathcal{B}_1^\ddagger}$ and $\boxed{\mathcal{B}_2 \sqsupseteq \mathcal{B}_2^\ddagger}$, $\boxed{\text{for some } \mathcal{B}_1^\ddagger,\mathcal{B}_2^\ddagger}$. □

Lemma 8. *Let R_1,R_2 be branching pomsets.*

(i) If $R_1 \sqsupseteq R_1'$ and $R_2 \sqsupseteq R_2'$ then $R_1 \parallel R_2 \sqsupseteq R_1' \parallel R_2'$.

(ii) If $R_1 \sqsupseteq R_1'$ and $R_2 \sqsupseteq R_2'$ then $R_1 + R_2 \sqsupseteq R_1'$, $R_1 + R_2 \sqsupseteq R_2'$ and $R_1 + R_2 \sqsupseteq R_1' + R_2'$.

(iii) If $R_1 \sqsupseteq R_1'$ and $R_2 \sqsupseteq R_2'$ then $R_1;R_2 \sqsupseteq R_1'\,;R_2'$.

(iv) If $R_1 \dagger R_2 \sqsupseteq R_3$ for $\dagger \in \{;,\parallel\}$ then $R_3 = R_1' \dagger R_2'$ for some $R_1 \sqsupseteq R_1'$ and $R_2 \sqsupseteq R_2'$.

(v) If $R_1 + R_2 \sqsupseteq R_3$ then either $R_3 = R_1'$ or $R_3 = R_2'$ or $R_3 = R_1' + R_2'$ for some $R_1 \sqsupseteq R_1', R_2 \sqsupseteq R_2'$.

Proof. Let $R_1 = \langle E_1, \leq_1, \lambda_1, \mathcal{B}_1\rangle$ with $\leq_1 = \leq_1{}^\star$ and similarly for R_2. By the rules in Fig. 7 $R_1' = R_1[\mathcal{B}_1'] = \langle E_1', \leq_1', \lambda_1', \mathcal{B}_1'\rangle$ for some $\mathcal{B}_1 \sqsupseteq \mathcal{B}_1'$ and analogously for R_2'.

(i) By the rules in Fig. 9 $R_1 \parallel R_2 = \langle E_1 \cup E_2, \leq_1 \cup \leq_2, \lambda_1 \cup \lambda_2, \mathcal{B}_1 \cup \mathcal{B}_2\rangle$. By Lemma 7(i) $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1' \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1' \cup \mathcal{B}_2'$. It follows that $R_1 \parallel R_2 \sqsupseteq (R_1 \parallel R_2)[\mathcal{B}_1' \cup \mathcal{B}_2'] = \langle E_1' \cup E_2', \leq_1' \cup \leq_2', \lambda_1' \cup \lambda_2', \mathcal{B}_1' \cup \mathcal{B}_2'\rangle = R_1' \parallel R_2'$.

(ii) By the rules in Fig. 9 $R_1 + R_2 = \langle E_1 \cup E_2, \leq_1 \cup \leq_2, \lambda_1 \cup \lambda_2, \{\{\mathcal{B}_1,\mathcal{B}_2\}\}\rangle$. By the rules in Fig. 7 $\{\{\mathcal{B}_1,\mathcal{B}_2\}\} \sqsupseteq \mathcal{B}_1'$. It follows that $R_1 + R_2 \sqsupseteq (R_1 + R_2)[\mathcal{B}_1'] = \langle E_1', \leq_1', \lambda_1', \mathcal{B}_1'\rangle = R_1'$. The case for R_2' is analogous. By the rules in Fig. 7 $\{\{\mathcal{B}_1,\mathcal{B}_2\}\} \sqsupseteq \{\{\mathcal{B}_1',\mathcal{B}_2'\}\}$. It follows that $R_1 + R_2 \sqsupseteq (R_1 + R_2)[\{\{\mathcal{B}_1',\mathcal{B}_2'\}\}] = R_1' + R_2'$.

(iii) By the rules in Fig. 7 $R_1\,;R_2 = \langle E_1 \cup E_2, \leq_1 \cup \leq_2 \cup \bigcup_{\mathsf{a}\in\mathcal{A}} E_{1_\mathsf{a}} \times E_{2_\mathsf{a}}, \lambda_1 \cup \lambda_2, \mathcal{B}_1 \cup \mathcal{B}_2\rangle$. By Lemma 7(i) $\mathcal{B}_1 \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1' \cup \mathcal{B}_2 \sqsupseteq \mathcal{B}_1' \cup \mathcal{B}_2'$. It follows that $R_1\,;R_2 \sqsupseteq (R_1\,;R_2)[\mathcal{B}_1' \cup \mathcal{B}_2'] = \langle E_1' \cup E_2', \leq_1' \cup \leq_2' \cup \bigcup_{\mathsf{a}\in\mathcal{A}} E'_{1_\mathsf{a}} \times E'_{2_\mathsf{a}}, \lambda_1' \cup \lambda_2', \mathcal{B}_1' \cup \mathcal{B}_2'\rangle = R_1'\,;R_2'$.

(iv) By the rules in Fig. 7 $(R_1 \dagger R_2).\mathcal{B} \sqsupseteq \mathcal{B}'$ and $R_3 = (R_1 \dagger R_2)[\mathcal{B}']$ for some $\mathcal{B}'$. By the rules in Fig. 9 $(R_1 \dagger R_2).\mathcal{B} = R_1.\mathcal{B} \cup R_2.\mathcal{B}$. It follows from Lemma 7(ii) that $\mathcal{B}' = \mathcal{B}_1' \cup \mathcal{B}_2'$ for some $R_1.\mathcal{B} \sqsupseteq \mathcal{B}_1'$ and $R_2.\mathcal{B} \sqsupseteq \mathcal{B}_2'$. By the rules in Fig. 7 $R_1 \sqsupseteq R_1[\mathcal{B}_1'] = R_1'$ and $R_2 \sqsupseteq R_2[\mathcal{B}_2'] = R_2'$. Finally, by the rules in Fig. 9 $R_3 = R_1' \dagger R_2'$.

(v) By the rules in Fig. 7 $(R_1 + R_2).\mathcal{B} \sqsupseteq \mathcal{B}'$ and $R_3 = (R_1 + R_2)[\mathcal{B}']$ for some $\mathcal{B}'$. By the rules in Fig. 9 $(R_1 + R_2).\mathcal{B} = \{\{\mathcal{B}_1, \mathcal{B}_2\}\}$. It follows from Lemma 7(iii) that either $\mathcal{B}' = \{\{\mathcal{B}_1', \mathcal{B}_2'\}\}$ or $\mathcal{B}' = \mathcal{B}_1'$ or $\mathcal{B}' = \mathcal{B}_2'$ for some $R_1.\mathcal{B} \sqsupseteq \mathcal{B}_1', R_2.\mathcal{B} \sqsupseteq \mathcal{B}_2'$. By the rules in Fig. 7 $R_1 \sqsupseteq R_1[\mathcal{B}_1'] = R_1'$ and $R_2 \sqsupseteq R_2[\mathcal{B}_2'] = R_2'$. If $\mathcal{B}' = \{\{\mathcal{B}_1', \mathcal{B}_2'\}\}$ then by the rules in Fig. 9 $R_3 = R_1' + R_2'$. The other two cases are analogous. □

Lemma 9. *Let R_1, R_2 be branching pomsets. Let e be an event.*

(i) If $R_1 \xrightarrow{\checkmark_e} R_1'$ then $R_1 \parallel R_2 \xrightarrow{\checkmark_e} R_1' \parallel R_2$.

(ii) If $R_1 \xrightarrow{\checkmark_e} R_1'$ then $R_1 + R_2 \xrightarrow{\checkmark_e} R_1'$.

(iii) If $R_1 \xrightarrow{\checkmark_e} R_1'$ then $R_1 ; R_2 \xrightarrow{\checkmark_e} R_1' ; R_2$.

Proof.

(i) By Lemma 8(i) $R_1 \parallel R_2 \sqsupseteq R_1' \parallel R_2$. Since $R_1 \xrightarrow{\checkmark_e} R_1'$, $e \in \text{a-min}(R_1')$ and then $e \in \text{a-min}(R_1' \parallel R_2)$.
Suppose that there exists some R'' such that $R_1 \parallel R_2 \sqsupseteq R'' \sqsupset R_1' \parallel R_2$ and $e \in \text{a-min}(R'')$. Then $R'' = R_1'' \parallel R_2$ for some $R_1 \sqsupseteq R_1'' \sqsupset R_1'$ such that $e \in \text{a-min}(R_1'')$, but this contradicts our premise that $R_1 \xrightarrow{\checkmark_e} R_1'$. We conclude that there exists no such R'' and then by the rules in Fig. 7 $R_1 \parallel R_2 \xrightarrow{\checkmark_e} R_1' \parallel R_2$.

(ii) By Lemma 8(ii) $R_1 + R_2 \sqsupseteq R_1'$. Since $R_1 \xrightarrow{\checkmark_e} R_1'$, $e \in \text{a-min}(R_1')$.
Suppose that there exists some R'' such that $R_1 + R_2 \sqsupseteq R'' \sqsupset R_1'$ and $e \in \text{a-min}(R'')$. For the latter to be true we have to resolve the outer choice $R_1 + R_2$, so $R_1 \sqsupseteq R'' \sqsupset R_1'$, but this contradicts our premise that $R_1 \xrightarrow{\checkmark_e} R_1'$. We conclude that there exists no such R'' and then by the rules in Fig. 7 $R_1 + R_2 \xrightarrow{\checkmark_e} R_1'$.

(iii) By Lemma 8(iii) $R_1 ; R_2 \sqsupseteq R_1' ; R_2$. Since $R_1 \xrightarrow{\checkmark_e} R_1'$, $e \in \text{a-min}(R_1')$ and then $e \in \text{a-min}(R_1' ; R_2)$.
Suppose that there exists some R'' such that $R_1 ; R_2 \sqsupseteq R'' \sqsupset R_1' ; R_2$ such that $e \in \text{a-min}(R'')$. Then $R'' = R_1'' ; R_2$ for some $R_1 \sqsupseteq R_1'' \sqsupset R_1'$ such that $e \in \text{a-min}(R_1'')$, but this contradicts our premise that $R_1 \xrightarrow{\checkmark_e} R_1'$. We conclude that there exists no such R'' and then by the rules in Fig. 7 $R_1 ; R_2 \xrightarrow{\checkmark_e} R_1' ; R_2$. □

Lemma 10. *Let c^* be a dependently guarded choreograpy and let $[\![c^*]\!] \xrightarrow{\ell} R'$ for some R'. Then $[\![c]\!] \xrightarrow{\ell} R''$ and $R' = R'' ; [\![c^*]\!]$ for some R''.*

Proof. Let $R = [\![c^*]\!]$. By Fig. 9 $R = [\![(R_1 ; R_2) + \mathbf{0}]\!]$ where $R_1 = [\![c]\!]$ and $R_2 = [\![c^*]\!]$. By Fig. 7 $R \xrightarrow{\checkmark_e} R_3$ such that $\lambda(e) = \ell$ and $R' = R_3 - e$. It follows that $R_1 ; R_2 \xrightarrow{\checkmark_e} R_3$. By Fig. 7 $R_1 ; R_2 \sqsupseteq R_3$ and then by Lemma 8(iv) $R_3 = R_1' ; R_2'$ for some $R_1 \sqsupseteq R_1'$ and $R_2 \sqsupseteq R_2'$, and either $e \in \text{a-min}(R_1')$ or $e \in \text{a-min}(R_2')$. If $e \in \text{a-min}(R_1')$ then $R_1 \xrightarrow{\checkmark_e} R_1'$ and by Lemma 9(iii) $R_1 ; R_2 \xrightarrow{\checkmark_e} R_1' ; R_2$. It follows that $R_3 = R_1' ; R_2$ and then $R' = (R_1' - e) ; [\![c^*]\!]$. Otherwise, i.e. if $e \in \text{a-min}(R_2')$, then $R_2 \xrightarrow{\checkmark_e} R_2'$ and by Lemma 1(b) $c \xrightarrow{\checkmark_\ell}$. However, since c^* is dependently guarded it follows that the subject of ℓ does not occur in c and then it also does not occur in c^*. As this is contradictory, e cannot be an event in R_2. □

On Composing Communicating Systems [*]

Franco Barbanera
Dept. of Mathematics and Computer Science, University of Catania (Italy)

Ivan Lanese
Focus Team, University of Bologna/INRIA (Italy)

Emilio Tuosto
Gran Sasso Science Institute (Italy)

Communication is an essential element of modern software, yet programming and analysing communicating systems are difficult tasks. A reason for this difficulty is the lack of compositional mechanisms that preserve relevant communication properties.

This problem has been recently addressed for the well-known model of *communicating systems*, that is sets of components consisting of finite-state machines capable of exchanging messages. The main idea of this approach is to take two systems, select a participant from each of them, and derive from those participants a pair of coupled gateways connecting the two systems. More precisely, a message directed to one of the gateways is forwarded to the gateway in the other system, which sends it to the other system. It has been shown that, under some suitable *compatibility* conditions between gateways, this composition mechanism preserves deadlock freedom for asynchronous as well as symmetric synchronous communications (where sender and receiver play the same part in determining which message to exchange).

This paper considers the case of *asymmetric synchronous communications* where senders decide independently which message should be exchanged. We show here that preservation of lock freedom requires sequentiality of gateways, while this is not needed for preservation of either deadlock freedom or strong lock freedom.

1 Introduction

Communication is an essential constitutive element of modern software due to the fact that applications are increasingly developed in distributed architectures (e.g., service-oriented architectures, microservices, cloud, etc.). In practice, APIs and libraries featuring different communication mechanisms are available for practically any programming language. At a theoretical level, several models have been used to study interactions between systems (e.g., process algebras, transition systems, Petri nets, logical frameworks, etc.).

Reasoning and developing communicating systems are difficult endeavours. The so-called *business logic*, necessary to determine *what* has to be communicated, needs to be complemented with the so-called *application level protocol* specifying *how* information spreads across a system. Conceptual and programming errors may occur in the realisation of application level protocols. For instance, it may happen that some components in a system are prevented to communicate because all the expected partners terminated their execution (deadlock). Other typical errors occur when a system is not lock-free, that is when some components cannot progress because all their partners are perpetually involved in other

[*] Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, by the MIUR project PRIN 2017FTXR7S “IT-MaTTerS” (Methods and Tools for Trustworthy Smart Systems) and by the Progetto di Ateneo “Piaceri”. The authors have also been partially supported by INdAM as members of GNCS (Gruppo Nazionale per il Calcolo Scientifico). The authors thanks the reviewers for their helpful comments and also Mariangiola Dezani for her support.

communications. A source of these problems is when a system can evolve in different ways depending on some conditions and components have inconsistent "views" of the state of the system. If this happens, some components may reach a state no longer "compatible" with the state of their partners and therefore communications cannot take place as expected.

We illustrate these problems with some simple examples for deadlock freedom (similar examples may be given for lock freedom). Suppose we want to model a client-server system where clients' requests are acknowledged either with an answer or with "unknown" from servers. Due to its popularity, we choose CCS [16] to introduce this scenario, so take the following agents:

$$\mathsf{C} = \bar{\mathsf{r}}.\mathsf{a} + \bar{\mathsf{r}}.\mathsf{u} \qquad \text{and} \qquad \mathsf{D} = \mathsf{r}.\bar{\mathsf{a}} + \mathsf{r}.\bar{\mathsf{u}} \tag{1}$$

where ports r, a, and u are respectively used to commmunicate requests, answers, and unknowns. (Recall that in CCS $\mathsf{a}.\_$ and $\_ + \_$ represent respectively prefix and non-deterministic choice.) The common interpretation of agents in (1) is that x and $\bar{\mathsf{x}}$ respectively represent an input and an output on port x. It is a simple observation that the system $\mathsf{C} \mid \mathsf{D}$ where C and D run in parallel can evolve to e.g., the deadlock state $\mathsf{a} \mid \bar{\mathsf{u}}$ where each party is waiting for the other to progress. The problem is that the choice of what communication should happen after a request is taken independently by C and D instead of letting D to take the decision and drive C "on the right" branch. This is attempted in the next version:

$$\mathsf{C} = \bar{\mathsf{r}}.(\mathsf{a} + \mathsf{u}) \qquad \text{and} \qquad \mathsf{D} = \mathsf{r}.(\bar{\mathsf{a}} + \bar{\mathsf{u}}) \tag{2}$$

A key difference with the agents in (1) is that the server D in (2) decides what to reply to the client C, which becomes aware of the choice through the interaction with D after the request has been made. Let us now assume that the server D acts as a proxy to another server, say D'. When D cannot return an answer to the client it interacts with D' on port p. Answers are sent directly to the client if D' can compute them, otherwise D' returns an unknown on port u' to D which forwards it to the client. This is modelled by the agents

$$\mathsf{D} = \mathsf{r}.(\bar{\mathsf{a}} + \bar{\mathsf{p}}.\mathsf{u}'.\bar{\mathsf{u}}) \qquad \text{and} \qquad \mathsf{D}' = \mathsf{p}.(\bar{\mathsf{a}} + \bar{\mathsf{u}'}) \tag{3}$$

Note that this change is completely transparent to agent C, which in fact stays as in (2). It is now more difficult to ascertain if these choices may lead to a deadlock since the decision of D may involve also D'. Indeed, the parallel composition of agents in (3) may deadlock because, when C and D interact on port a, D' hangs on port p and, likewise, if C and D' interact on port a then D hangs on port u'.

A reason for this difficulty is that it is hard to define compositional mechanisms that preserve relevant communication properties such as deadlock or lock freedom. Recently, an approach to the composition of concurrent and distributed systems has been proposed in [2, 3] for the well-known model of systems of *communicating finite-state machines* (CFSMs) [11], that is sets of finite-state automata capable of exchanging messages. The compositional mechanism is based on the idea that two given systems, say S and S', are composed by transforming two CFSMs, say H in S and K in S', into "coupled forwarders". Basically, each message that H receives from a machine in S is forwarded to K and vice versa. It has been shown that, under suitable *compatibility* conditions between H and K, this composition mechanism preserves deadlock freedom for asynchronous as well as symmetric synchronous communications (where sender and receiver play the same part in determining which message to exchange). The compatibility condition identified in [2, 3] consists in exhibiting essentially dual behaviours: gateway H is able to receive a message whenever gateway K is willing to send one and vice versa. As observed in [4], a remarkable feature of such an approach is that it enables the composition of systems originally designed as *closed* systems. As far as two compatible machines can be found, any two systems can be composed by transforming as hinted above the compatible machines.

The results in [2, 3] are developed in the asynchronous semantics of CFSMs. These results have been transferred in [5] to a setting where CFSMs communicate synchronously much like as the communication mechanisms considered for instance in process algebras like CCS, ACP, etc. This model assumes a perfect symmetry between sender and receiver in synchronous communications. Let us again discuss this with an example. Consider the agents

$$\mathsf{T} = \bar{\mathsf{a}}.\mathsf{P} + \bar{\mathsf{b}}.\mathsf{Q} \qquad \text{and} \qquad \mathsf{R} = \mathsf{a}.\mathsf{P}' + \mathsf{b}.\mathsf{Q}' \tag{4}$$

According to the standard semantics of CCS [16], system $(\mathsf{T} \mid \mathsf{R}) \setminus \{\mathsf{a},\mathsf{b}\}$ has two possible evolutions:

$$(\mathsf{T} \mid \mathsf{R}) \setminus \{\mathsf{a},\mathsf{b}\} \xrightarrow{\tau} \mathsf{P} \mid \mathsf{P}' \qquad \text{and} \qquad (\mathsf{T} \mid \mathsf{R}) \setminus \{\mathsf{a},\mathsf{b}\} \xrightarrow{\tau} \mathsf{Q} \mid \mathsf{Q}'$$

namely, either both T and R opt for the “leftmost” branch (synchronising on a) or they both choose the “rightmost” one (synchronising on b). (Recall that in CCS $\_ \setminus X$ is the hiding of ports in the set X and that τ represents an internal action.) This means that, the resolution of the choice is implicit in the communication mechanism: a branch is taken as soon as T and R synchronise on the corresponding port. Intuitively, no distinction is made between sender and receiver (formally they are indeed interchangeable); this implies that the communication mechanism is at the very core of choice resolution [5].

Interestingly, for synchronous communications, an alternative interpretation is actually possible where this perfect symmetry is not assumed so that sender and receiver play different roles in choice resolution while still relying on synchronous communication. Let us explain this interpretation using again CCS. Consider a variant of CCS where outputs must be enabled before being fired. One could formally specify that with the following reduction rules:

$$\bar{\mathsf{a}}.\mathsf{P} + \mathsf{P}' \xrightarrow{\tau} \bar{\bar{\mathsf{a}}}.\mathsf{P} \qquad \text{and} \qquad \bar{\bar{\mathsf{a}}}.\mathsf{P} \mid (\mathsf{a}.\mathsf{Q} + \mathsf{Q}') \xrightarrow{\tau} \mathsf{P}|\mathsf{Q} \tag{5}$$

whereby the leftmost rule *chooses* one of the possible outputs of the sender (the chosen output is marked by the double bar in our notation) and the rightmost rule actually synchronises sender and receiver. This semantics is an abstract model of *asymmetric* communications (used e.g., in [7, 17]), where silent steps taken using the left rule model some internal computation of the sender to decide what to communicate to the partner. In other words, now the choice is entirely resolved on “one side” while the communication is a mere interaction of complementary actions, the output and the input. This asymmetry, at the core of asynchronous communication, can therefore also carry for synchronous communication.

It is worth observing that asymmetric communications abstract a rather common programming pattern where sending components may choose the output to execute *depending* on some internal computation. For instance, elaborating on the proxy scenario in (3), D could decide to directly send unknowns to normal clients while reserving the use of D' only for “privileged” clients.

Contributions. This paper transfers the composition by gateway mechanism of [2, 3] to the case of asymmetric synchronous communication of CFSMs. The main technical results are that, in the asymmetric case, gateway composition

- preserves deadlock freedom (as well as a strong version of lock freedom) provided that systems are composable (the relation of compatibility – one of the requirements for systems to be composable – in the present paper is less restrictive than the one used in [5]);

- preserves lock freedom if systems are composable and gateways are *sequential*, namely each state has at most one outgoing transition.

Interestingly, preservation of deadlock freedom can be guaranteed under milder conditions than in the symmetric case. In fact, sequentiality of gateways is not necessary to preserve deadlock freedom in the asymmetric case, while it is in the symmetric one.

Structure of the paper. Section 2 introduces systems of (asymmetric synchronous) CFSMs, related notions and communication properties. Composition by gateways is introduced and discussed in Section 3 together with the compatibility relation. Section 4 discusses the issues that prevent gateway composition to preserve communication properties. Section 5 is devoted to the preservation of communication properties. Conclusions, related and future work are discussed in Section 6.

2 Background

Communicating Finite State Machines (CFSMs) [11] are Finite State Automata (FSAs) where transitions are labelled by communications. We recall basic notions on FSAs.

A *finite state automaton* (FSA) is a tuple $A = \langle \mathscr{S}, q_0, \mathscr{L}, \rightarrow \rangle$ where

- $\mathscr{S}$ is a finite set of states (ranged over by lowercase italic Latin letters);
- $q_0 \in \mathscr{S}$ is the *initial state*;
- $\mathscr{L}$ is a finite set of labels
- $\rightarrow \subseteq \mathscr{S} \times (\mathscr{L} \cup \{\tau\}) \times \mathscr{S}$ is a set of transitions.

Hereafter, we let λ range over $\mathscr{L} \cup \{\tau\}$ when it is immaterial to specify the set of labels or it is understood. We use the usual notation $q_1 \xrightarrow{\lambda} q_2$ for the transition $(q_1, \lambda, q_2) \in \rightarrow$, and $q_1 \rightarrow q_2$ when there exists a label λ such that $q_1 \xrightarrow{\lambda} q_2$. Let $\_\cdot\_$ be the concatenation operation on labels and write $p \xrightarrow{\pi} q$ where $\pi = \lambda_1 \cdot \lambda_2 \cdot \ldots \cdot \lambda_n$ whenever $p \xrightarrow{\lambda_1} p_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} p_n = q$. We let $\pi, \psi, \ldots$ range over $\mathscr{L}^\star$ (i.e., sequences of labels) and define the set of *reachable states in A from q* as

$$\mathscr{R}(A, q) = \{ p \mid \text{there is } \pi \in \mathscr{L}^\star \text{ such that } q \xrightarrow{\pi} p \}$$

The set of *reachable states in A* is $\mathscr{R}(A) = \mathscr{R}(A, q_0)$. For succinctness, $q \xrightarrow{\lambda} q' \in A$ means that the transition belongs to (the set of transitions of) A; likewise, $q \in A$ means that q belongs to the states of A. We say that $q \xrightarrow{\lambda} q'$ is an *outgoing* (resp. *incoming*) transition of q (resp. q'). Since we use FSAs to formalise communicating systems, accepting states are disregarded (as also done in [11]).

We now define systems of CFSMs, by adapting the definitions in [11] to our context. Let $\mathfrak{P}$ be a set of *participants* (or *roles*, ranged over by A, B, etc.) and $\mathscr{M}$ a set of *messages* (ranged over by m, n, etc.). We take $\mathfrak{P}$ and $\mathscr{M}$ disjoint. An *output label* is written as A B!m and represents the willingness of A to send message m to B; likewise, an *input label* is written as A B?m and represents the willingness of B to receive message m from A. The *subjects* of an output label A B!m and of an input label A B?m are A and B, respectively.

Definition 2.1 (CFSMs)**.** *A* communicating finite-state machine *(CFSM) is an FSA M with labels in the set $\mathscr{L}_{act} \cup \{\tau\}$, where*

$$\mathscr{L}_{act} = \{\mathsf{A\,B!m}, \mathsf{A\,B?m} \mid \mathsf{A} \neq \mathsf{B} \in \mathfrak{P}, \mathsf{m} \in \mathscr{M}\}$$

and, for any transition $p \xrightarrow{\lambda} q$,

- *if λ is an output label then $p \neq q$ and p has exactly one incoming transition, and such transition is labelled by τ;*
- *if $\lambda = \tau$ then $p \neq q$ and q has exactly one outgoing transition, and such transition is labelled by an output label.*

A state of M is

- terminal, *if it has no outgoing transition; we define $\mathsf{T}(M) = \{ p \in M \mid p \text{ is terminal in } M \}$*
- sending, *if it is not terminal and all its outgoing transitions have output labels*
- receiving, *if it is not terminal and all its outgoing transitions have input labels*
- mixed, *if it has a silent outgoing transition and an outgoing transition with an input label.*

A CFSM is A-local *if all its non τ transitions have subject A.*

Unlike in [5], the transitions of our CFSMs can also be labelled by the silent action τ. Definition 2.1 can be looked at as the CFSM counterpart of the τC contracts described in [6]. Imposing the no-mixed state condition on our CFSM, turns them into the communicating automata counterpart of the processes (contracts) called "session behaviours"[1] in e.g., [10, 1, 7]. These processes are in turn the process counterpart of (binary) session types [13]. As we shall see below (and also shown in [3] and [5]), the absence of mixed states is necessary in order to get the preservation of properties by composition. As a matter of fact, we could drop the conditions related to τ-transitions in case a transition like $p \xrightarrow{\mathsf{XY!z}} q$ is the only outgoing transition from p, namely when no actual choice of output actions is possible in p. We however prefer to avoid this distinction for several reasons:

- firstly, our uniform treatment of transitions allows us to immediately adapt definitions in a more abstract setting;
- secondly, uniformity allows us to simplify some technicalities.

Said that, all proofs in the present paper could easily be adapted to the above mentioned alternative definition of CFSM.

Definition 2.2 (Communicating systems)**.** *A* (communicating) system over $\mathscr{P}$ *is a map* $\mathsf{S} = (M_{\mathsf{A}})_{\mathsf{A} \in \mathscr{P}}$ *assigning an A-local CFSM M_{A} to each participant $\mathsf{A} \in \mathscr{P}$ where $\mathscr{P} \subseteq \mathfrak{P}$ is finite and any participant occurring in a transition of M_{A} is in $\mathscr{P}$.*

Note that Definition 2.2 requires that any input or output label does refer to participants belonging to the system itself. In other words, Definition 2.2 restricts to *closed* systems.

We now define the synchronous semantics of communicating systems, which is itself an FSA (differently from the asynchronous case, where the set of states can be infinite). Hereafter, we write $f[x \mapsto y]$ for the update of the function f in a point x of its domain with the value y. Also, $\mathrm{dom}(f)$ denotes the domain of the function f.

Definition 2.3 (Asymmetric synchronisations)**.** *Let S be a communicating system. A* configuration of S *is a map $s = (q_{\mathsf{A}})_{\mathsf{A} \in \mathrm{dom}(\mathsf{S})}$ assigning a* local state *$q_{\mathsf{A}} \in \mathsf{S}(\mathsf{A})$ to each $\mathsf{A} \in \mathrm{dom}(\mathsf{S})$.*

The asymmetric synchronisations *of S is the FSA*

$$[\![\mathsf{S}]\!] = \langle \mathscr{S}, s_0, \mathscr{L}_{int} \cup \{ \tau \}, \to \rangle \qquad \textit{where}$$

- *$\mathscr{S}$ is the set of synchronous configurations of S, as defined above;*

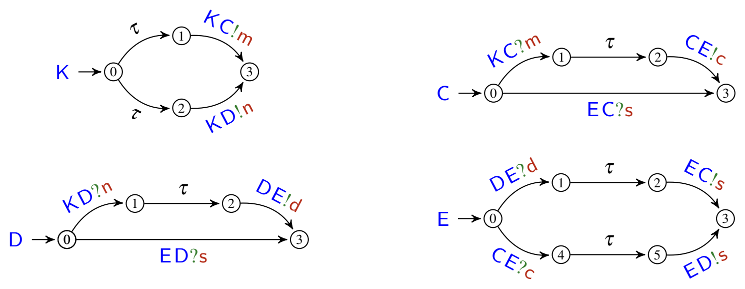
[1]Actually different variations of this name are used in the listed references.

- $s_0 = (q_{0\mathsf{A}})_{\mathsf{A}\in\mathrm{dom}(\mathsf{S})} \in \mathscr{S}$ *is the* initial *configuration where, for each* $\mathsf{A} \in \mathrm{dom}(\mathsf{S})$, $q_{0\mathsf{A}}$ *is the initial state of* $\mathsf{S}(\mathsf{A})$*;*
- $\mathscr{L}_{int} = \{\mathsf{A}\to\mathsf{B}\colon \mathsf{m} \mid \mathsf{A} \neq \mathsf{B} \in \mathfrak{P}$ *and* $\mathsf{m} \in \mathscr{M}\}$ *is a set of interaction labels;*
- $s \xrightarrow{\mathsf{A}\to\mathsf{B}\colon \mathsf{m}} s[\mathsf{A} \mapsto q, \mathsf{B} \mapsto q'] \in [\![\mathsf{S}]\!]$ *if* $s(\mathsf{A}) \xrightarrow{\mathsf{A\,B!m}} q \in \mathsf{S}(\mathsf{A})$ *and* $s(\mathsf{B}) \xrightarrow{\mathsf{A\,B?m}} q' \in \mathsf{S}(\mathsf{B})$*;*
- $s \xrightarrow{\tau} s[\mathsf{A} \mapsto q] \in [\![\mathsf{S}]\!]$ *if* $s(\mathsf{A}) \xrightarrow{\tau} q \in \mathsf{S}(\mathsf{A})$*;*

Configuration s enables A in S *if* $s(\mathsf{A})$ *has at least an outgoing transition.*

As expected, an interaction $\mathsf{A}\to\mathsf{B}\colon \mathsf{m}$ occurs when A performs an output $\mathsf{A\,B!m}$ (which has been previously chosen) and B the corresponding input $\mathsf{A\,B?m}$.

Example 2.4. *Let us consider the communicating system* $\mathsf{S} = (M_{\mathsf{X}})_{\mathsf{X}\in\{\mathsf{K},\mathsf{C},\mathsf{D},\mathsf{E}\}}$*, where*



A sequence of transitions of $[\![\mathsf{S}]\!]$ *out of* s_0 *is, according to Definition 2.3,*

$$\begin{array}{rlcl} s_0 = (0_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) & \xrightarrow{\tau} (1_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) & \xrightarrow{\mathsf{K}\to\mathsf{C}\colon \mathsf{m}} & (3_{\mathsf{K}}, 1_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) \\ & \xrightarrow{\tau} (3_{\mathsf{K}}, 2_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) & \xrightarrow{\mathsf{C}\to\mathsf{E}\colon \mathsf{c}} & (3_{\mathsf{K}}, 3_{\mathsf{C}}, 0_{\mathsf{D}}, 4_{\mathsf{E}}) \\ & \xrightarrow{\tau} (3_{\mathsf{K}}, 3_{\mathsf{C}}, 0_{\mathsf{D}}, 5_{\mathsf{E}}) & \xrightarrow{\mathsf{E}\to\mathsf{D}\colon \mathsf{s}} & (3_{\mathsf{K}}, 3_{\mathsf{C}}, 3_{\mathsf{D}}, 3_{\mathsf{E}}) \end{array}$$

◇

The symmetric synchronisation in [5] for systems without τ-transitions can be readily obtained from the above definition by disregarding the clause for the τ-transitions.

In the following, $\mathsf{ptp}(\tau) = \emptyset$ and $\mathsf{ptp}(\mathsf{A}\to\mathsf{B}\colon \mathsf{m}) = \mathsf{ptp}(\mathsf{A\,B!m}) = \mathsf{ptp}(\mathsf{A\,B?m}) = \{\mathsf{A}, \mathsf{B}\}$ and, for a sequence $\pi = \lambda_1 \cdot \cdots \cdot \lambda_n$, we let $\mathsf{ptp}(\pi) = \cup_{1\le i\le n}\mathsf{ptp}(\lambda_i)$.

As discussed in Section 1, we shall study the preservation of communication properties under composition. We shall consider the following relevant properties: deadlock freedom, lock freedom and strong lock freedom. The definitions below adapt the ones in [12] to a synchronous setting (as done also in [15, 18, 5]).

Definition 2.5 (Communication properties)**.** *Let* S *be a communicating system on* $\mathscr{P}$*. We say that a participant* $\mathsf{A} \in \mathscr{P}$ *is* involved *in a run* $s \xrightarrow{\lambda_1} s_1 \ldots \xrightarrow{\lambda_n} s_n$ *of* S *if there is* $1 \le i \le n$ *such that either* $\mathsf{A} \in ptp(\lambda_i)$ *or* $\lambda_i = \tau$, $s_i(\mathsf{A}) \xrightarrow{\tau} q$ *in* $\mathsf{S}(\mathsf{A})$*, and* $s_{i+1} = s_i[\mathsf{A} \mapsto q]$*.*

Deadlock freedom *A configuration* $s \in \mathscr{R}([\![\mathsf{S}]\!])$ *is a* deadlock *if*

- *s has no outgoing transitions in* $[\![\mathsf{S}]\!]$ *and*
- *there exists* $\mathsf{A} \in \mathscr{P}$ *such that* $s(\mathsf{A})$ *enables* A *in* S.

A system is deadlock-free *if none of its configurations is a deadlock.*

Lock freedom *Let* $\mathsf{A} \in \mathscr{P}$*. A configuration* $s \in \mathscr{R}([\![\mathsf{S}]\!])$ *is a* lock *for* A *if*

- *$s(\mathsf{A})$ has outgoing transitions; and*
- *A is not involved in any run from s.*

A system is lock-free *if none of its configurations is a lock for any of its participants.*

Strong lock freedom *System S is* strongly lock-free for $\mathsf{A} \in \mathscr{P}$ *if for each $s \in \mathscr{R}([\![\mathsf{S}]\!])$ enabling A in S then A is involved in all maximal sequences from s.*

A system is strongly lock free *if it is strongly lock free for each of its participants.*

Proposition 2.6. *1. Lock-freedom implies deadlock-freedom;*

2. Strong lock freedom implies lock freedom.

Example 2.7. *Let us consider the system S of Example 2.4. The only other maximal transition sequence in $[\![\mathsf{S}]\!]$ out of s_0, besides the one described in Example 2.4, is*

$$\begin{array}{llll} s_0 = (0_{\mathsf{K}},0_{\mathsf{C}},0_{\mathsf{D}},0_{\mathsf{E}}) & \xrightarrow{\tau} & (2_{\mathsf{K}},0_{\mathsf{C}},0_{\mathsf{D}},0_{\mathsf{E}}) & \xrightarrow{\mathsf{K}\rightarrow\mathsf{D}\colon\mathsf{n}} & (3_{\mathsf{K}},0_{\mathsf{C}},1_{\mathsf{D}},0_{\mathsf{E}}) \\ & \xrightarrow{\tau} & (3_{\mathsf{K}},0_{\mathsf{C}},2_{\mathsf{D}},0_{\mathsf{E}}) & \xrightarrow{\mathsf{D}\rightarrow\mathsf{E}\colon\mathsf{d}} & (3_{\mathsf{K}},0_{\mathsf{C}},3_{\mathsf{D}},1_{\mathsf{E}}) \\ & \xrightarrow{\tau} & (3_{\mathsf{K}},0_{\mathsf{C}},3_{\mathsf{D}},2_{\mathsf{E}}) & \xrightarrow{\mathsf{E}\rightarrow\mathsf{C}\colon\mathsf{s}} & (3_{\mathsf{K}},3_{\mathsf{C}},3_{\mathsf{D}},3_{\mathsf{E}}) \end{array}$$

These two sequences are both maximal and contain all the elements of $\mathscr{R}([\![\mathsf{S}]\!])$. By the above observations it is possible to check S to be strongly lock free. ◇

3 Composition via Gateways

This section discusses composition of systems of CFSMs via gateways, as introduced in [2, 3], and studies its properties under asymmetric synchronisation. The main idea is that two systems of CFSMs, say S_1 and S_2, can be composed by transforming one participant in each of them into gateways connected to each other.

3.1 Building gateways

Let us call H the selected participant in S_1 and K the one in S_2. The gateways for H and K are connected to each other and act as forwarders: each message sent to the gateway for H by a participant from the original system S_1 is now forwarded to the gateway for K, that in turn forwards it to the same participant to which K sent it in the original system S_2. The dual will happen to messages that the gateway for K receives from S_2. A main advantage of this approach is that no extension of the CFSM model is needed to transform systems of CFSMs, which are normally closed systems, into open systems that can be composed. Another advantage is that the composition is fully transparent to all participants different from H and K.

We will now define composition via gateways on systems of CFSMs, following the intuition above.

Definition 3.1 (Gateway)**.** *Given a H-local CFSM M and a participant K, the* gateway of M towards K *is the CFSM* $\mathsf{gw}(M,\mathsf{K})$ *obtained by replacing in M*

- *each pair of consecutive transitions $p \xrightarrow{\tau} q \xrightarrow{\mathsf{HA!m}} r$ with*

$$p \xrightarrow{\mathsf{KH?m}} p' \xrightarrow{\tau} q \xrightarrow{\mathsf{HA!m}} r \qquad \text{for some fresh state } p' \tag{6}$$

- *each transition $p \xrightarrow{\mathsf{AH?m}} r$ with*

$$p \xrightarrow{\mathsf{AH?m}} p' \xrightarrow{\tau} p'' \xrightarrow{\mathsf{HK!m}} r \qquad \text{for some fresh states } p' \text{ and } p'' \tag{7}$$

We shall call external *the states like p and r and* internal *the states like p', p'' and q.*

Note that gateways execute "segments" of the form described in (6) and (7) in the above definition. Also, by very construction, we have the following

Fact 3.2. *Given a* H*-local CFSM M and a participant* K*, each state of* $\mathsf{gw}(M,\mathsf{K})$ *has at most one incoming or outgoing τ transition.*

We compose systems with disjoint participants through two of them, say H and K, by taking all the participants of the original systems but H and K, whereas H and K are replaced by their respective gateways.

Given two functions f and g such that $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$, we let $f + g$ denote the function behaving as function f on $\mathrm{dom}(f)$ and as function g on $\mathrm{dom}(g)$.

Definition 3.3 (System composition)**.** *Let S_1 and S_2 be two systems with disjoint domains. The* composition of S_1 and S_2 via $\mathsf{H} \in \mathrm{dom}(\mathsf{S}_1)$ and $\mathsf{K} \in \mathrm{dom}(\mathsf{S}_2)$ *is defined as*

$$\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2 = \mathsf{S}_1[\mathsf{H} \mapsto \mathsf{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})] + \mathsf{S}_2[\mathsf{K} \mapsto \mathsf{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})]$$

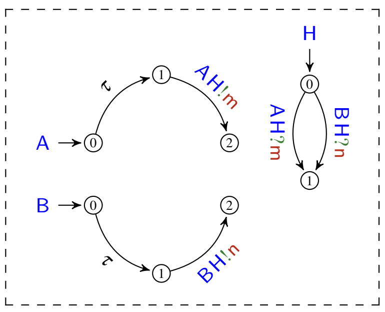
(Note that $\mathrm{dom}(\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2) = \mathrm{dom}(\mathsf{S}_1) \cup \mathrm{dom}(\mathsf{S}_2)$*.)*

We remark again that, by the above approach for composition, we do not actually need to formalise the notion of *open* system. In fact any closed system can be looked at as open by choosing (according to the current needs) two suitable participants in the "to-be-connected" systems and transforming them into two forwarders.

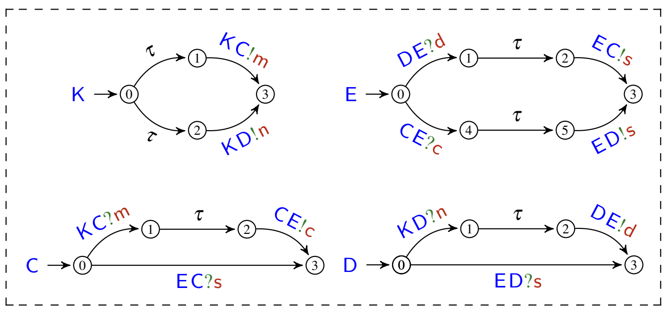
We also note that the notion of composition above is structural: a corresponding notion of behavioural composition has been studied in [4] in a context of multiparty session types [14], that is with symmetric synchronous interactions.

Example 3.4. *Let us take the following two communicating systems.*

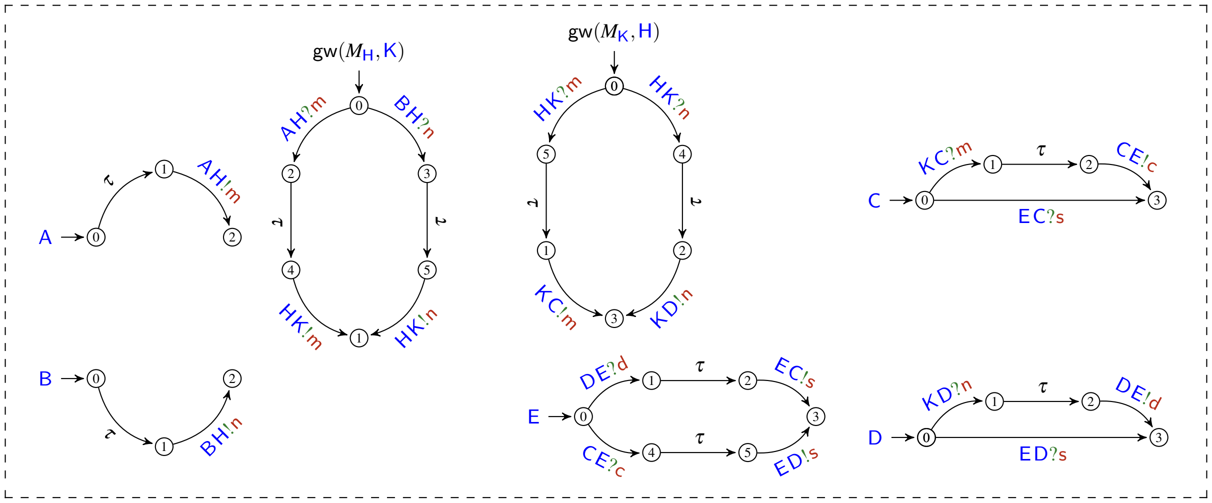
$\mathsf{S}_1 =$



$\mathsf{S}_2 =$



The system $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *is*



Note that the CFSMs A, B, C, D, *and* E *remain unchanged.* ◇

3.2 Compatibility

A few simple auxiliary notions are useful. Let $\mathscr{L}_{\text{i/o}} = \{\, ?\mathsf{m}, !\mathsf{m} \mid \mathsf{m} \in \mathscr{M} \,\}$ and define the functions

$$\mathsf{io} : \mathscr{L}_{\text{act}} \to \mathscr{L}_{\text{i/o}} \qquad \text{and} \qquad \overline{(\cdot)} : \mathscr{L}_{\text{i/o}} \to \mathscr{L}_{\text{i/o}}$$

by the following clauses

$$\mathsf{io}(\mathsf{A\,B?m}) = ?\mathsf{m} \qquad \mathsf{io}(\mathsf{A\,B!m}) = !\mathsf{m} \qquad \text{and} \qquad \overline{?\mathsf{m}} = !\mathsf{m} \qquad \overline{!\mathsf{m}} = ?\mathsf{m}$$

which extend to CFSMs in the obvious way: given a CFSM $M = \langle \mathscr{S}, q_0, \mathscr{L}_{\text{act}}, \to \rangle$, we define $\mathsf{io}(M) = \langle \mathscr{S}, q_0, \mathscr{L}_{\text{i/o}}, \to' \rangle$ where $\to' = \{ q \xrightarrow{\mathsf{io}(\lambda)} q' \mid q \xrightarrow{\lambda} q' \in M, \lambda \in \mathscr{L}_{\text{act}} \} \cup \{ q \xrightarrow{\tau} q' \mid q \xrightarrow{\tau} q' \in M \}$ and likewise for $\overline{M}$.

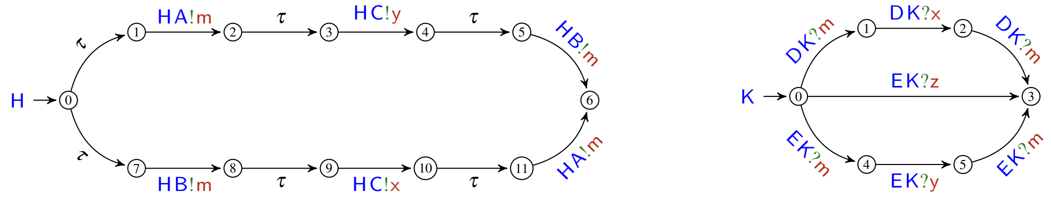
Informally, two CFSMs M_1 and M_2 are *compatible* if each output of M_1 has a corresponding input in M_2 and vice versa once the identities of communicating partners are blurred away.

Definition 3.5 (Compatibility)**.** *Let M and M' be two FSAs on $\mathscr{L}_{i/o}$. An* io-correspondence *is a relation R between states of M and those of M' such that whenever $(q,q') \in R$:*

- *$q \in \mathsf{T}(M)$ if, and only if, $q' \in \mathsf{T}(M')$ (cf. Definition 2.1)*
- *if $q \xrightarrow{!\mathsf{m}} r \in M$ then there is $q' \xrightarrow{?\mathsf{m}} r' \in M'$ such that $(r,r') \in R$*
- *if $q' \xrightarrow{!\mathsf{m}} r' \in M'$ then there is $q \xrightarrow{?\mathsf{m}} r \in M$ such that $(r,r') \in R$*
- *if $q \xrightarrow{\tau} r \in M$ then $(r,q') \in R$*
- *if $q' \xrightarrow{\tau} r' \in M'$ then $(q,r') \in R$*

Two CFSMs M and M' are compatible *(in symbols $M \asymp M'$) if there is an io-correspondence relating the initial states of* $\mathsf{io}(M_1)$ *and* $\mathsf{io}(M')$.

Example 3.6 (Compatibility)**.** *The machines* H *and* K *of Example 3.4 are compatible. For a more complex example, consider the following CFSMs*



The above H *and* K *are compatible. Apart for* τ *actions preceding them,* H *can only perform output actions, whereas* K *can only perform input actions. By disregarding the names of the receivers in the actions of* H*, and of the senders in those of* K*, any output action after its corresponding* τ *can find a matching input in* K*. The vice versa does not hold, since none of the possible output actions that can occur after a* τ *from* 0 *(i.e. the outputs from* 1 *and* 7 *in* H*) can actually match the input action* EK?z *from* 0 *in* K*. Such a possibility is in fact allowed by our definition of compatibility.* ◇

Definition 3.5 transfers the notion of compatibility given in [4] for processes in multiparty sessions. Also, Definition 3.5 differs from the notions of compatibility in [5] and in [2, 3] which are defined as bisimulations and do not involve τ-transitions.

Definition 3.7. *An* A*-local CFSM M is:*

1. ?-deterministic *if* $p \xrightarrow{\mathsf{XA?m}} q$ *and* $p \xrightarrow{\mathsf{YA?m}} r \in M$ *implies* $q = r$*;*
2. !-deterministic *if* $p \xrightarrow{\tau} \xrightarrow{\mathsf{AX!m}} q$ *and* $p \xrightarrow{\tau} \xrightarrow{\mathsf{AY!m}} r \in M$ *implies* $q = r$*;*
3. ?!-deterministic *if it is both ?-deterministic and !-deterministic;*

A non-terminal state $q \in M$ is *asymmetric sending* (resp. *receiving*) if all its outgoing transitions have τ (resp. receiving) labels; q is a *asymmetric mixed* state if it is neither asymmetric sending nor receiving.

Example 3.8. *Machine* H *and* K *in Example 3.6 are, respectively non !-deterministic and non ?-deterministic. In particular, conditions (2) and (1) of Definition 3.7 fail for, respectively, state* 0 *of* H *and state* 0 *of* K.

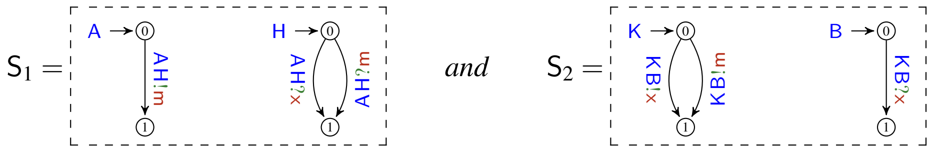
We require a stronger condition then compatibity for two systems to be composable.

Definition 3.9 ((H,K)-composability)**.** *Two systems* S_1 *and* S_2 *with disjoint domains are* (H,K)-composable *if* $\mathsf{H} \in \mathrm{dom}(\mathsf{S}_1)$ *and* $\mathsf{K} \in \mathrm{dom}(\mathsf{S}_2)$ *are two compatible ?!-deterministic machines with no asymmetric mixed states.*

4 Composition Related Issues

It is known that under symmetric synchronisation composition spoils deadlock-freedom; this is shown by the example below, borrowed from [5].

Example 4.1 (Deadlock-freedom preservation fails under symmetric synchronisation)**.** *Take the following systems*



Clearly, S_1 *and* S_2 *are* (H,K)*-composable and deadlock-free, yet their composition* $\mathsf{S} = \mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *has a deadlock. In fact, when the gateway for* K *receives* m*, it tries to synchronise with participant* B *on message* m *while* B *is waiting only for* x*. For* S_2 *in isolation, this is not a deadlock, since* B *and* K *synchronise on* x *under the symmetric semantics.* ◇

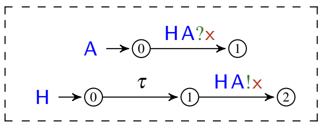
Notice that the counterexample of Example 4.1 does not apply in an asynchronous setting. Indeed, the second system could deadlock due to the fact that K could send m without synchronising with B. Likewise, the counterexample of Example 4.1 does not apply in our asymmetric setting. Even if communication is still synchronous, the τ-transitions introduced to resolve internal choices (i.e., those prefixing outputs) allow S_2 to reach a deadlock configuration by choosing the τ-transition leading to the output $\mathsf{K}\,\mathsf{B}!\mathsf{m}$.

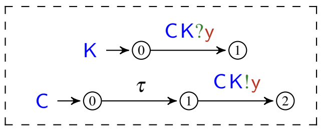
Now, one may think that analogously to what happens in [2, 3], if two systems are (H,K)-composable and deadlock-free then their composition is deadlock-free too.

In Section 5 we shall prove that in our setting lock-freedom is preserved by composition, without any further condition beside (H,K)-composability. Before doing that, we give examples showing the necessity of our conditions for deadlock freedom preservation.

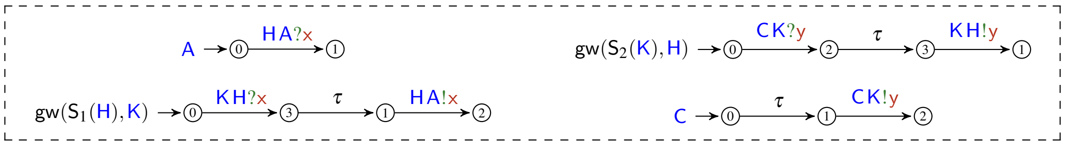
Let us begin with compatibility. Properties cannot be preserved under composition without compatibility, as shown in the next example.

Example 4.2 (Lack of compatibility spoils deadlock freedom preservation)**.** *Let us consider the following communicating systems.*

$\mathsf{S}_1 =$ 

and $\mathsf{S}_2 =$ 

These systems are trivially deadlock free. However, H *and* K *are not compatible, since there is no corresponding input in* K *for the output from* H*. The composition of* S_1 *and* S_2 *via* H *and* K *yields*



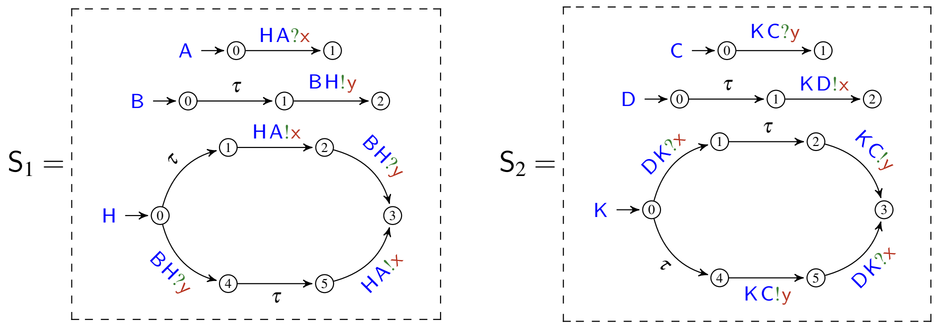
Starting from the initial configuration of $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$*, the following transitions are possible in* $[\![\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2]\!]$

$$(0_{\mathsf{A}},0_{\mathsf{H}},0_{\mathsf{K}},0_{\mathsf{C}}) \xrightarrow{\tau} (0_{\mathsf{A}},0_{\mathsf{H}},0_{\mathsf{K}},1_{\mathsf{C}}) \xrightarrow{\mathsf{C}\longrightarrow\mathsf{K}:\,\mathsf{y}} (0_{\mathsf{A}},0_{\mathsf{H}},2_{\mathsf{K}},2_{\mathsf{C}}) \xrightarrow{\tau} (0_{\mathsf{A}},0_{\mathsf{H}},3_{\mathsf{K}},2_{\mathsf{C}}) \not\rightarrow$$

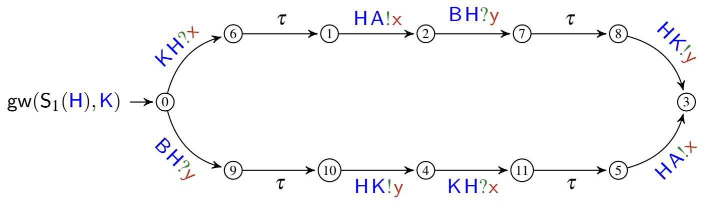
where $(0_{\mathsf{A}},0_{\mathsf{H}},3_{\mathsf{K}},2_{\mathsf{C}})$ *is a deadlock configuration for* $[\![\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2]\!]$ *since* K *wishes to send* y *to* H*, which is instead waiting for message* x*.* ◇

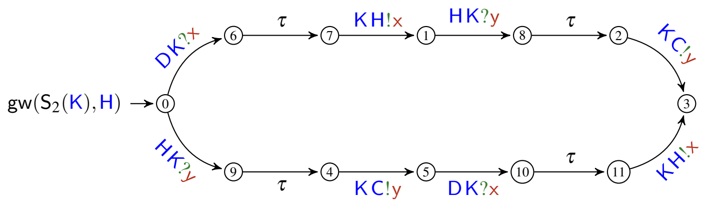
The following example casts in our setting an example given in [3] for the asynchronous semantics; this example illustrates that asymmetric mixed states must be avoided to preserve properties under composition.

Example 4.3 (Asymmetric mixed-states spoil deadlock freedom preservation)**.** *Let* S_1 *and* S_2 *be*



Notice that the initial states are asymmetric mixed and that H *and* K *are compatible. The gateways we obtain are*



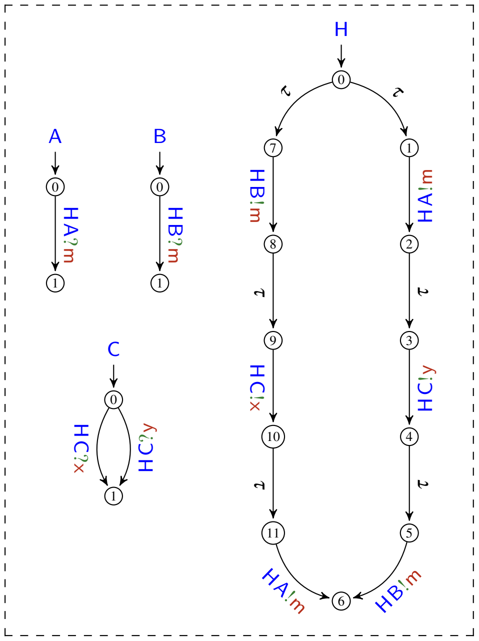


The composed system $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *deadlocks when* $\mathsf{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$ *receives from* B *while* $\mathsf{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ *receives from* D *since both gateways reach an output state (respectively states* 10 *and* 7*).* ◇

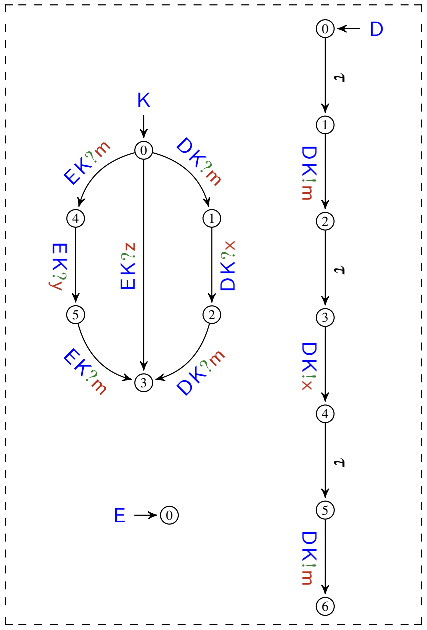
The following examples show that, as asymmetric mixed states, !?-nondeterminism is problematic too. Let us first take two deadlock free systems.

Example 4.4. *The two systems*

$\mathsf{S}_1 =$



and $\mathsf{S}_2 =$



are deadlock free.

The deadlock freedom of S_1 *follows from the fact that, from its initial configuration* $(0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 0_{\mathsf{H}})$, S_1 *can only branch over the two* τ*-transitions of* H *reaching either of the following configurations*

$$(0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 7_{\mathsf{H}}) \qquad \textit{or} \qquad (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 1_{\mathsf{H}})$$

From the former (resp. latter) configuration S_1 *can only reach configurations where* H *synchronises with* C *and then with* A *(resp.* B*). In either case* S_1 *reaches the terminal configuration* $(1_{\mathsf{A}}, 1_{\mathsf{B}}, 1_{\mathsf{C}}, 6_{\mathsf{H}})$.

Let us now have a look at S_2. *Firstly note that* E *cannot synchronise since it is already terminated; hence, the only possible transitions of* S_2 *must involve* K *and* D *only. We therefore have that*

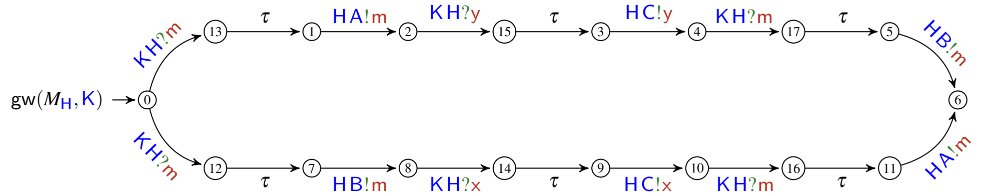
$$\begin{aligned}(0_{\mathsf{K}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) &\xrightarrow{\tau} (0_{\mathsf{K}}, 1_{\mathsf{D}}, 0_{\mathsf{E}}) \xrightarrow{\mathsf{D} \to \mathsf{K} \colon \mathsf{m}} (1_{\mathsf{K}}, 2_{\mathsf{D}}, 0_{\mathsf{E}}) \xrightarrow{\tau} (1_{\mathsf{K}}, 3_{\mathsf{D}}, 0_{\mathsf{E}}) \xrightarrow{\mathsf{D} \to \mathsf{K} \colon \mathsf{x}} (2_{\mathsf{K}}, 4_{\mathsf{D}}, 0_{\mathsf{E}}) \\ &\xrightarrow{\tau} (2_{\mathsf{K}}, 5_{\mathsf{D}}, 0_{\mathsf{E}}) \xrightarrow{\mathsf{D} \to \mathsf{K} \colon \mathsf{m}} (3_{\mathsf{K}}, 6_{\mathsf{D}}, 0_{\mathsf{E}})\end{aligned}$$

is the only possible execution from the initial configuration $(0_{\mathsf{K}}, 0_{\mathsf{D}}, 0_{\mathsf{E}})$ *of* S_2*, leading to the terminal configuration* $(3_{\mathsf{K}}, 6_{\mathsf{D}}, 0_{\mathsf{E}})$.

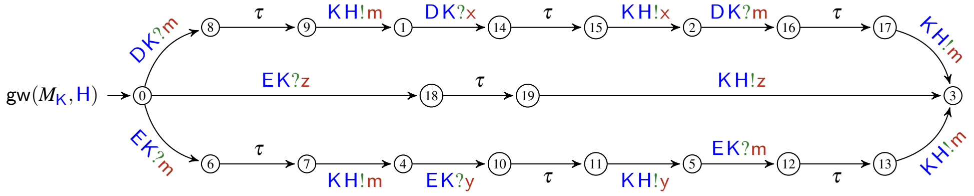
◇

The next example shows that the compositions of the systems S_1 and S_2 in Example 4.4 can deadlock.

Example 4.5 (?!-determinism is necessary)**.** *The CFSMs* H *and* K *in Example 4.4 are compatible as seen* *in Example 3.6. Hence, we can build the composed system* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *through the gateways*



and



Now, from the initial configuration $s_0 = (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 0_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 0_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 0_{\mathsf{D}}, 0_{\mathsf{E}})$ *of* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *we have the following run*

$$
\begin{aligned}
s_0 &\xrightarrow{\tau} (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 0_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 0_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 1_{\mathsf{D}}, 0_{\mathsf{E}}) \\
&\xrightarrow{\mathsf{D}\rightarrow\mathsf{K}:\ \mathsf{m}} \xrightarrow{\tau} (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 0_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 9_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 2_{\mathsf{D}}, 0_{\mathsf{E}}) \\
&\xrightarrow{\mathsf{K}\rightarrow\mathsf{H}:\ \mathsf{m}} \xrightarrow{\tau} (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 13_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 1_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 3_{\mathsf{D}}, 0_{\mathsf{E}}) && (8) \\
&\xrightarrow{\mathsf{D}\rightarrow\mathsf{K}:\ \mathsf{x}} (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 13_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 14_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 4_{\mathsf{D}}, 0_{\mathsf{E}}) \\
&\xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 1_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 15_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 5_{\mathsf{D}}, 0_{\mathsf{E}}) \\
&\xrightarrow{\mathsf{H}\rightarrow\mathsf{A}:\ \mathsf{m}} (1_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{C}}, 2_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 15_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 5_{\mathsf{D}}, 0_{\mathsf{E}}) && (9)
\end{aligned}
$$

where the τ*-transition of* D *enables the synchronisation of* $\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})$ *and* D *with label* $\mathsf{D}\rightarrow\mathsf{K}:\ \mathsf{m}$ *that leads the gateway in state* 9 *after its* τ*-transition from state* 8*. Now, the two gateways can communicate and exchange message* m*. Due to* ?!*-nondeterminism of* S_1*, from state* 0 $\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})$ *can move either to state* 12 *or to state* 13*. Fatally, transition (8) leads to a deadlock: after* $\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})$ *and* D *synchronise to exchange message* x *the system goes into a configuration from where* $\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})$ *forwards* m *to* A *and reaches the last configuration (9). This is a deadlock for* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$*, since none of the CFSMs can do a* τ*-transitions, the only enabled output action is from* $\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})$ *which tries to send message* x *to* $\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})$*; however,* $\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})$ *can only receive message* y *from* K *and hence these actions cannot synchronise.* $\diamond$

5 Preserving Properties by Composition

Composition via gateways does not ensure the preservation of communication properties. We provide below sufficient conditions for this to happen. Recall that (H,K)-composability requires absence of asymmetric mixed states and ?!-determinism.

Theorem 5.1 (Deadlock freedom preservation)**.** *Let* S_1 *and* S_2 *be two* (H,K)*-composable and deadlock-free systems. Then the composed system* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *is deadlock-free.*

Proof sketch. The proof relies on the fact that the reachable configurations of $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ can be projected on reachable configurations of S_1 and S_2. This implies that a deadlock in $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ corresponds to a deadlock in S_1 or in S_2. See the appendix for the detailed proof. □

Example 5.2. *We can infer deadlock-freedom of the system $\mathsf{S} = \mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ of Example 3.4 by the result above, since S_1 and S_1 are (H,K)-composable and deadlock-free.*

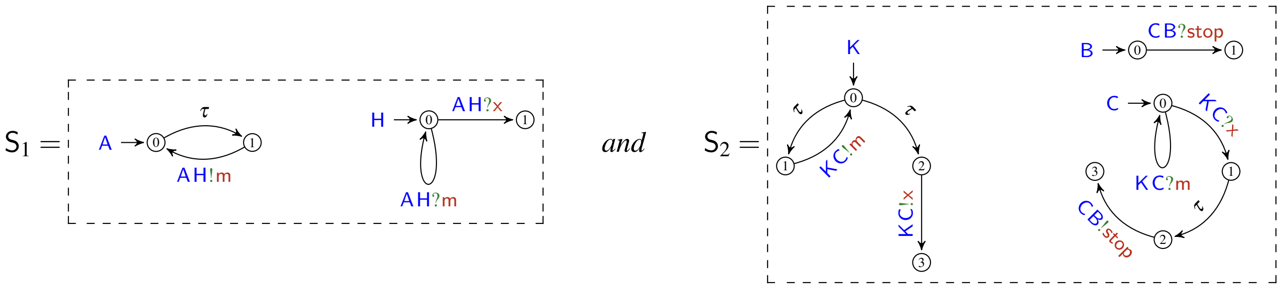
Somehow surprisingly, in the symmetric case preservation of deadlock freedom requires stricter conditions on gateways than in the asymmetric case. In fact, in the asymmetric case, deadlock freedom preservation requires only absence of asymmetric mixed states and ?!-determinism while the symmetric case requires the (stronger) condition of *sequentiality*.

Definition 5.3 (Sequential CFSM)**.** *A CFSM is* sequential *if each of its states has at most one outgoing transition. A participant A of a system S is sequential if $\mathsf{S}(\mathsf{A})$ is so.*

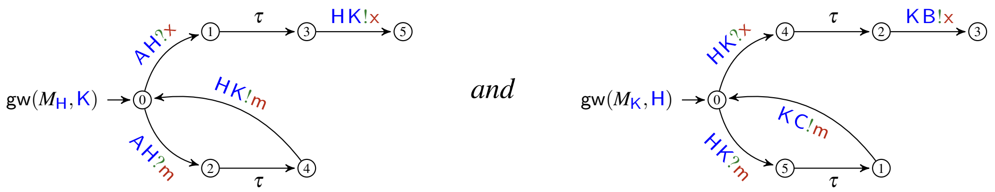
As we will see (cf. Theorem 5.5), sequentiality is necessary to preserve lock-freedom also in the asymmetric case. We note that sequentiality implies absence of asymmetric mixed states and ?!-determinism, while the converse does not hold.

As mentioned before, the property of lock freedom is not preserved in general by composition, as shown by the following example.

Example 5.4 (Composability does not preserve lock-freedom)**.** *Take the communicating systems*



Note that both S_1 and S_2 are lock-free and that H and K are compatible. The gateways are



Hence, the composed system $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ is non lock-free because e.g. the configuration

$$s = (0_{\mathsf{A}}, 0_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 0_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 0_{\mathsf{B}}, 0_{\mathsf{C}})$$

is a lock for B, since the only outgoing transition from 0_{B} could be fired only in case the transition $\mathsf{C}\,\mathsf{B}!\mathsf{stop}$ is enabled. However, this is impossible since $\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})$ forwards only message m; hence, the run (which does not involve B)

$$s \xrightarrow{\tau} \xrightarrow{\mathsf{A}\rightarrow\mathsf{H}:\ \mathsf{m}} (0_{\mathsf{A}}, 2_{\mathsf{gw}(M_{\mathsf{H}},\mathsf{K})}, 0_{\mathsf{gw}(M_{\mathsf{K}},\mathsf{H})}, 0_{\mathsf{B}}, 0_{\mathsf{C}}) \xrightarrow{\tau} \xrightarrow{\mathsf{A}\rightarrow\mathsf{H}:\ \mathsf{m}} s\cdots$$

is perpetually executed. ◇

We show that the problem of Example 5.4 cannot happen in case we restrict to sequential gateways, as done for deadlock freedom in the symmetric case (cf. [5]). As usual, $f\,|_X$ denotes the restriction of a function f on a subset X of its domain.

Theorem 5.5 (Lock-freedom preservation)**.** *Let* S_1 *and* S_2 *be two* (H,K)*-composable and lock-free systems with* H *and* K *sequential. Then the composed system* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *is lock-free.*

Proof sketch. The proof goes as the one of Theorem 5.1 noticing that we have to reconstruct "backward" the sequence of interactions. This exploits sequentiality and lock-freedom of S_1 and S_2 in order to guarantee the reconstruction when we "cross" the two composed systems through the gateways. □

We turn now our attention to strong lock-freedom. In this case, as for deadlock freedom, (H,K)-composability suffices for preservation by composition; we shall see that this is not the case for lock freedom preservation.

Theorem 5.6 (Strong lock freedom preservation)**.** *Let* S_1 *and* S_2 *be two* (H,K)*-composable and strongly lock free systems. Then the composed system* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *is strongly lock free.*

Proof sketch. The proof is similar to the one of Theorem 5.5 but for the use of strong lock freedom of S_1 and S_2 instead of their deadlock freedom. □

6 Conclusions and Future Work

We introduce an asymmetric synchronous semantics of communicating systems which breaks the symmetry between senders and receivers. In fact, our semantics decouples communication from choice resolution as in standard semantics of communicating systems (and other models). We then adapted the gateway composition mechanism defined in [2, 3] to our asymmetric semantics and gave conditions for the preservation of some communication properties under this notion of composition.

An approach related to ours is the framework of [8, 9] based on *contract automata* where transitions express "requests" and "offers" among participants. The composition mechanism is based on "trimming" a product of contract automata according to relevant *agreement* properties. This yields controllers that preserve deadlocks. Contract automata do not consider asymmetric synchronous semantics. Our composition mechanism does not introduce orchestrators which, under some conditions, can be avoided also for contract automata [8, 9].

Modular approaches to the development of concurrent systems can be exploited even for systems designed using formalisms intrinsically dealing with *closed* systems. Indeed, given two systems, any two components – one per system – exhibiting *compatible* behaviours can be replaced by two coupled forwarders (gateways) connecting the systems, as investigated initially in [2, 3] for an asynchronous interaction model. The investigation on the composition-by-gateways technique was shifted in [5] towards synchronous symmetric interactions. In the present paper we pushed a step forward such an investigation, by considering *asymmetric* synchronous interactions. Interestingly, deadlock freedom preservation in the synchronous asymmetric case we consider does not require sequentiality of gateways, like in the asynchronous case, and differently from the synchronous symmetric case. Notably, sequentiality is needed here for lock-freedom preservation, but not for strong-lock freedom preservation.

While the path of investigation above is quite homogeneous, the different analyses present some methodological differences. For instance, [5] considers also another form of composition, where one single gateway (interacting with both the composed systems) is used. On the other side, [5] focused only on deadlocks, disregarding other properties we consider. A first item of future research consist in filling the bits missing due to the mismatches above.

A more challenging direction for future work is looking for refined composition mechanisms in order to get preservation of relevant properties under weaker conditions.

References

[1] F. Barbanera and U. de'Liguoro. Sub-behaviour relations for session-based client/server systems. *MSCS*, 25(6):1339–1381, 2015.

[2] F. Barbanera, U. de'Liguoro, and R. Hennicker. Global types for open systems. In M. Bartoletti and S. Knight, editors, *ICE*, volume 279 of *EPTCS*, pages 4–20, 2018.

[3] F. Barbanera, U. de'Liguoro, and R. Hennicker. Connecting open systems of communicating finite state machines. *JLAMP*, 109, 2019.

[4] F. Barbanera, M. Dezani-Ciancaglini, I. Lanese, and E. Tuosto. Composition and decomposition of multiparty sessions. *JLAMP*, 2020. Submitted.

[5] F. Barbanera, I. Lanese, and E. Tuosto. Composing communicating systems, synchronously. In T. Margaria and B. Steffen, editors, *ISoLA 2020*, volume 12476 of *LNCS*, pages 39–59. Springer, 2020.

[6] M. Bartoletti, T. Cimoli, and R. Zunino. Compliance in behavioural contracts: A brief survey. In C. Bodei, G. L. Ferrari, and C. Priami, editors, *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, volume 9465 of *Lecture Notes in Computer Science*, pages 103–121. Springer, 2015.

[7] M. Bartoletti, A. Scalas, and R. Zunino. A semantic deconstruction of session types. In P. Baldan and D. Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 402–418. Springer, 2014.

[8] D. Basile, P. Degano, G. L. Ferrari, and E. Tuosto. Playing with our CAT and communication-centric applications. In E. Albert and I. Lanese, editors, *FORTE*, volume 9688 of *LNCS*, pages 62–73. Springer, 2016.

[9] D. Basile, M. H. ter Beek, and R. Pugliese. Synthesis of orchestrations and choreographies: Bridging the gap between supervisory control and coordination of services. *LMCS*, 16(2), 2020.

[10] G. Bernardi and M. Hennessy. Modelling session types using contracts. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1941–1946, New York, NY, USA, 2012. ACM.

[11] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[12] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *I&C*, 202(2):166–190, 2005.

[13] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[14] H. Hüttel et al. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[15] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.

[16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, Berlin, 1980.

[17] L. Padovani. Contract-based discovery of web services modulo simple orchestrators. *Theoretical Computer Science*, 411:3328–3347, 2010.

[18] E. Tuosto and R. Guanciale. Semantics of global view of choreographies. *JLAMP*, 95:17–40, 2018.

A Proofs for Section 3 (Composition via Gateways)

Given a configuration of the composition of systems S_1 and S_2 we can retrieve the configurations of the two subsystems by taking only the states of participants in S_1 and S_2 while avoiding, for the gateways, to take the fresh states introduced by the gateway construction. Indeed, we shall prove in Proposition B.3 that for each $s \in \mathscr{R}([\![\mathsf{S}]\!])$, we have ${}_{\mathsf{H}}\rfloor s \in \mathscr{R}([\![\mathsf{S}_1]\!])$ and $s\lfloor_{\mathsf{K}} \in \mathscr{R}([\![\mathsf{S}_2]\!])$.

Definition A.1 (Configuration projections)**.** *Let s be a configuration of a composed system $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ and $M = \mathsf{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$. The* left-projection of s on S_1 *is the map ${}_{\mathsf{H}}\rfloor s$ defined on* $\mathsf{dom}(\mathsf{S}_1)$ *by*

$$
{}_{\mathsf{H}}\rfloor s : \mathsf{A} \mapsto \begin{cases} q, & \text{if } s(\mathsf{A}) \notin \mathsf{S}_1(\mathsf{A}) \text{ and there is } q \xrightarrow{\mathsf{K\,H?m}} s(\mathsf{A}) \in M \text{ or } q \xrightarrow{\mathsf{K\,H?m}} \xrightarrow{\tau} s(\mathsf{A}) \in M \\ r, & \text{if } s(\mathsf{A}) \notin \mathsf{S}_1(\mathsf{A}) \text{ and there is } s(\mathsf{A}) \xrightarrow{\mathsf{H\,K!m}} r \in M \text{ or } s(\mathsf{A}) \xrightarrow{\tau} \xrightarrow{\mathsf{H\,K!m}} r \in M \\ s(\mathsf{A}), & \text{otherwise} \end{cases}
$$

The definition of right-projection $s\lfloor_{\mathsf{K}}$ *is analogous.*

Proposition A.2. *Left- and right-projections are well-defined.*

Proof. It is enough to show that ${}_{\mathsf{H}}\rfloor\cdot$ and $\cdot\lfloor_{\mathsf{K}}$ uniquely assign a state to fresh states because on non-fresh stats both functions act as the identify map. This follows since, by construction, each state introduced by our gateway construction has unique successor and predecessor. □

Intuitively, only H is aware of an input from K when H is in the internal state reached after an input from K; hence to have a coherent configuration we take the state of H before the input. If instead H is in an internal state corresponding to an output to K, then other participants in S_1 know that the message has been sent; hence to have a coherent configuration we take the state of H after the send. (A similar intuition applies to $s\lfloor_{\mathsf{K}}$.)

Example A.3. *Let $s = (2_{\mathsf{A}}, 0_{\mathsf{B}}, 1_{\mathsf{H}}, 5_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}})$; and $\mathsf{S} = \mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ be the system of Example 3.4. Then, $s \in \mathscr{R}([\![\mathsf{S}]\!])$, namely s is reachable in S. In fact*

$$
\begin{aligned}
s_0 = (0_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{H}}, 0_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) \quad & \xrightarrow{\tau} \quad (1_{\mathsf{A}}, 0_{\mathsf{B}}, 0_{\mathsf{H}}, 0_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) \\
& \xrightarrow{\mathsf{A}\to\mathsf{H}:\,\mathsf{m}} \quad (2_{\mathsf{A}}, 0_{\mathsf{B}}, 2_{\mathsf{H}}, 0_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) \\
& \xrightarrow{\tau} \quad (2_{\mathsf{A}}, 0_{\mathsf{B}}, 4_{\mathsf{H}}, 0_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}}) \\
& \xrightarrow{\mathsf{H}\to\mathsf{K}:\,\mathsf{m}} \quad (2_{\mathsf{A}}, 0_{\mathsf{B}}, 1_{\mathsf{H}}, 5_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}})
\end{aligned}
$$

The projections of s on, respectively, S_1 and S_2 are ${}_{\mathsf{H}}\rfloor s = (2_{\mathsf{A}}, 0_{\mathsf{B}}, 1_{\mathsf{H}})$ and $s\lfloor_{\mathsf{K}} = (0_{\mathsf{K}}, 0_{\mathsf{C}}, 0_{\mathsf{D}}, 0_{\mathsf{E}})$. ◇

B Proofs for Section 4 (Composition Related Issues)

Let M be an H-local CFSM and $\mathsf{K} \in \mathscr{P} \setminus \{\mathsf{H}\}$ be a participant not occurring in M. Function nof maps the states of $\mathsf{gw}(M,\mathsf{K})$ to the states of M as follows:

$$
\mathsf{nof}_{\mathsf{H},\mathsf{K}}(M,q) = \begin{cases} p, & \text{if } p \xrightarrow{\lambda} q \in \mathsf{gw}(M,\mathsf{K}) \text{ and } \lambda \text{ input label with } \mathsf{K} \notin \mathsf{ptp}(\lambda) \\ p, & \text{if } p \xrightarrow{\lambda} p' \xrightarrow{\tau} q \in \mathsf{gw}(M,\mathsf{K}) \text{ and } \lambda \text{ input label with } \mathsf{K} \in \mathsf{ptp}(\lambda) \\ r, & \text{if } q \xrightarrow{\lambda} r \in \mathsf{gw}(M,\mathsf{K}) \text{ and } \lambda \text{ output label with } \mathsf{K} \notin \mathsf{ptp}(\lambda) \\ r, & \text{if } q \xrightarrow{\tau} q' \xrightarrow{\lambda} r \in \mathsf{gw}(M,\mathsf{K}) \text{ and } \lambda \text{ output label with } \mathsf{K} \in \mathsf{ptp}(\lambda) \\ q, & \text{if } q \text{ is a state of } M \end{cases}
$$

Note that the first four clause imply that q is a fresh state of $\mathsf{gw}(M, \mathsf{K})$.

Lemma B.1. *Function* nof *is well-defined.*

Proof. Let M be an H-local CFSM and $\mathsf{K} \in \mathscr{P} \setminus \{\mathsf{H}\}$. We have to check only internal states as the restriction of nof to the states of M is the identity by definition. If q is an internal state of $\mathsf{gw}(M, \mathsf{K})$, by definition of $\mathsf{gw}(M, \mathsf{K})$, there is a unique q' such that either $q' \xrightarrow{\mathsf{A\,H?m}} q \in \mathsf{gw}(M, \mathsf{K})$ or $q \xrightarrow{\mathsf{H\,A!m}} q' \in \mathsf{gw}(M, \mathsf{K})$. □

Example B.2. *For* $\mathsf{S}_1 {}^{\mathsf{H}\leftrightarrow\mathsf{K}} \mathsf{S}_2$ *of Example 3.4,* $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), 0) = 0$ *and* $\mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), 2) = 0$. ◇

Function nof is similar to configuration projection when considering CFSMs in isolation, with a main difference: when e.g., $\mathsf{gw}(M, \mathsf{K})$ in state q receives from a partner in its own system going to some fresh state p' with a τ-transition to p'', nof maps both p' and p'' to p (unlike configuration projection ${}_{\mathsf{H}}\rfloor\_$ which maps q to the successor of p''). This represents the fact that the other system, and K in particular, are oblivious of the transition. In fact, function nof is designed to establish a correspondence with the other system as shown by the next proposition.

Proposition B.3. *Let* $\mathsf{S} = \mathsf{S}_1 {}^{\mathsf{H}\leftrightarrow\mathsf{K}} \mathsf{S}_2$ *be the composition of two* (H, K)*-composable systems* S_1 *and* S_2*. If* $s \in \mathscr{R}(\llbracket \mathsf{S} \rrbracket)$ *then* ${}_{\mathsf{H}}\rfloor s \in \mathscr{R}(\llbracket \mathsf{S}_1 \rrbracket)$*,* $s\lfloor_{\mathsf{K}} \in \mathscr{R}(\llbracket \mathsf{S}_2 \rrbracket)$*, and* $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) \asymp \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$.

Proof. Let s_0 be the initial configuration of S and

$$s_0 \xrightarrow{\lambda_1} s_1 \cdots s_{n-1} \xrightarrow{\lambda_n} s_n = s \tag{10}$$

a run reaching s from s_0. We proceed by induction on n.

If $n = 0$ the thesis is immediate by observing that

- ${}_{\mathsf{H}}\rfloor s \in \mathscr{R}(\llbracket \mathsf{S}_1 \rrbracket)$ and $s\lfloor_{\mathsf{K}} \in \mathscr{R}(\llbracket \mathsf{S}_2 \rrbracket)$ because left- and right-projections are the initial configurations of S_1 and S_2 respectively, and
- $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H}))$ and $\mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$ are the initial states of $\mathsf{S}_1(\mathsf{H})$ and $\mathsf{S}_2(\mathsf{K})$ respectively which are compatible by hypothesis.

Let $n > 0$ and assume that the statement holds for all configurations reachable from s_0 in less the n transitions. We have that either none of H and K are involved in λ_n or that at least one of them is. In the former case, without loss of generality, assume that the interacting participants, say A and B are both in S_1. Then, by construction (cf. Definition A.1), $s\lfloor_{\mathsf{K}} = s_{n-1}\lfloor_{\mathsf{K}}$ and by inductive hypothesis $s_{n-1}\lfloor_{\mathsf{K}} \in \mathscr{R}(\llbracket \mathsf{S}_2 \rrbracket)$. Moreover, ${}_{\mathsf{H}}\rfloor s$ equals ${}_{\mathsf{H}}\rfloor s_{n-1}$ but for the local states of A and B; hence ${}_{\mathsf{H}}\rfloor s_{n-1} \in \mathscr{R}(\llbracket \mathsf{S}_1 \rrbracket)$ by the semantics of communicating systems (cf. Definition 2.3). Also,

$$\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) = ({}_{\mathsf{H}}\rfloor s)(\mathsf{H}) = ({}_{\mathsf{H}}\rfloor s_{n-1})(\mathsf{H}) \asymp (s_{n-1}\lfloor_{\mathsf{K}})(\mathsf{K}) = (s\lfloor_{\mathsf{K}})(\mathsf{K}) = \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$$

because the equalities above hold by the definition of asymmetric synchronisation (cf. Definition 2.3) and the compatibility relation holds by the inductive hypothesis. So, let us assume that at least one between H and K is involved in the last transition reaching s and proceed by case analysis on λ_n.

$\boxed{\lambda_n = \tau}$ We consider only the case where H is involved since the case where K is involved is symmetric. By our gateway construction (cf. Definition 3.1) only one of the following two cases are possible for the transitions in $\mathsf{gw}(\mathsf{S}_1(\mathsf{H}), \mathsf{K})$:

$$p \xrightarrow{\mathsf{K\,H?m}} s_{n-1}(\mathsf{H}) \xrightarrow{\tau} q \xrightarrow{\mathsf{H\,A!m}} r \tag{11}$$

$$q \xrightarrow{\mathsf{A\,H?m}} s_{n-1}(\mathsf{H}) \xrightarrow{\tau} p \xrightarrow{\mathsf{H\,K!m}} r \tag{12}$$

for some participant A of S_1 and states p, q, r of $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$. Cases (11) and (12) respectively correspond to have the transitions

$$p \xrightarrow{\tau} q \xrightarrow{\mathsf{HA!m}} r \qquad \text{and} \qquad q \xrightarrow{\mathsf{AH?m}} r$$

in the machine $\mathsf{S}_1(\mathsf{H})$.

In case (11) the last transition in the run (10) must therefore be preceded by a transition, say the i-th one, where the machines $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$ and $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ have exchanged message m. Hence, we have that $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ has a transition $s_i(\mathsf{K}) \xrightarrow{\mathsf{KH!m}} s_{i+1}(\mathsf{K})$ with $s_{n-1}(\mathsf{H}) \asymp s_{i+1}(\mathsf{K})$ since (by inductive hypothesis) $s_i(\mathsf{H}) = p = \mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_i(\mathsf{H})) \asymp \mathrm{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_i(\mathsf{K}))$ and both gateways are ?!-deterministic. We now observe that either $s(\mathsf{K}) = s_{i+1}(\mathsf{K})$ or it is a fresh state that $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ reaches after having received a message from a participant of S_2 (possibly followed by a τ transition) between the i-th and the last interactions in (10). We have that ${}_{\mathsf{H}}\rfloor s = {}_{\mathsf{H}}\rfloor s_{n-1}[\mathsf{H} \mapsto p]$, hence ${}_{\mathsf{H}}\rfloor s \in \mathscr{R}([\![\mathsf{S}_1]\!])$ by inductive hypothesis; also, $s\lfloor_{\mathsf{K}} \in \mathscr{R}([\![\mathsf{S}_2]\!])$ by the inductive hypothesis since $s\lfloor_{\mathsf{K}} = s_{n-1}\lfloor_{\mathsf{K}}$ because K is not involved in λ_n. Finally, in both cases the last part of the thesis immediately follows by observing that $\mathrm{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K})) = \mathrm{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_i(\mathsf{K}))$ by definition and $\mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) = p$.

In case (12), firstly note that by construction $\mathsf{A} \neq \mathsf{H}$ (cf. Definition 2.3). Then $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$ has the transition $s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{AH?m}} s(\mathsf{H})$ which corresponds to an input transition $s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{AH?m}} r$ where $r = {}_{\mathsf{H}}\rfloor s(\mathsf{H}) = \mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_{n-1}(\mathsf{H})) = \mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H}))$ in $\mathsf{S}_1(\mathsf{H})$. Hence, the thesis follows since $s\lfloor_{\mathsf{K}} = s_{n-1}\lfloor_{\mathsf{K}}$ because K is not involved in λ_n and $\mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) \asymp \mathrm{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K})) = \mathrm{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_{n-1}(\mathsf{K}))$ by inductive hypothesis.

$\boxed{\lambda_n = \mathsf{H} \to \mathsf{X} \colon \mathsf{m}}$ By construction, either $\mathsf{X} = \mathsf{K}$ or $\mathsf{X} \neq \mathsf{H}$ is a participant of S_1. In the former case, $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$ has the transition $s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{HK!m}} s(\mathsf{H})$ which corresponds to an input transition while $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ has the transition $s_{n-1}(\mathsf{K}) \xrightarrow{\mathsf{HK?m}} s(\mathsf{K})$ which corresponds to a sequence of transitions $p \xrightarrow{\tau} \xrightarrow{\mathsf{KB!m}} s(\mathsf{K})$ in $\mathsf{S}_2(\mathsf{K})$ for some participant $\mathsf{B} \neq \mathsf{K}$ in S_2. Then the thesis follows by the fact that ${}_{\mathsf{H}}\rfloor s = {}_{\mathsf{H}}\rfloor s_{n-1}[\mathsf{H} \mapsto s(\mathsf{H})]$ and $s\lfloor_{\mathsf{K}} = s_{n-1}\lfloor_{\mathsf{K}}[\mathsf{K} \mapsto p]$ by construction (cf. Definition A.1) and that, by inductive hypothesis, $\mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_{n-1}(\mathsf{H})) = q \asymp s(\mathsf{K}) = \mathrm{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_{n-1}(\mathsf{K}))$ and therefore, by ?!-determinism and the compatibility relation $s(\mathsf{H}) \asymp s(\mathsf{K})$.

Suppose now that X is a participant of S_1; note that by construction $\mathsf{X} \neq \mathsf{H}$ (cf. Definition 2.3). Since $s\lfloor_{\mathsf{K}} = s_{n-1}\lfloor_{\mathsf{K}}$, the inductive hypothesis immediately entails that $s\lfloor_{\mathsf{K}} \in \mathscr{R}([\![\mathsf{S}_2]\!])$.

We first show the reachability of left- and right-projections. The transition $s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{HX!m}} s(\mathsf{H})$ is in $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$ by construction and it corresponds to a pair of transitions $p \xrightarrow{\tau} s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{HX!m}} s(\mathsf{H})$ in $\mathsf{S}_1(\mathsf{H})$. We have that ${}_{\mathsf{H}}\rfloor s_{n-1} \in \mathscr{R}([\![\mathsf{S}_1]\!])$ (by inductive hypothesis) and since ${}_{\mathsf{H}}\rfloor s_{n-1}(\mathsf{H}) = p$ (by Definition A.1) we have ${}_{\mathsf{H}}\rfloor s_{n-1} \xrightarrow{\tau} \xrightarrow{\mathsf{H} \to \mathsf{X} \colon \mathsf{m}} {}_{\mathsf{H}}\rfloor s$ (by Definition 2.3).

We now show the compatibility condition. The last transition in the run (10) must be preceded by a transition, say the i-th one, where the machines $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}),\mathsf{K})$ and $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ have exchanged message m. Hence, we have that $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ has a transition $s_i(\mathsf{K}) \xrightarrow{\mathsf{KH!m}} s_{i+1}(\mathsf{K})$ with

$$\mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_{n-1}(\mathsf{H})) = s(\mathsf{H}) = \mathrm{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) \tag{13}$$

which hold by definition of $\mathrm{nof}(\_,\_)$. We now observe that either $s(\mathsf{K}) = s_{i+1}(\mathsf{K})$ or it is a fresh state that $\mathrm{gw}(\mathsf{S}_2(\mathsf{K}),\mathsf{H})$ reaches after having received a message from a participant of S_2 (possibly followed by a τ transition) between the i-th and the last interactions in (10). In both cases the $\mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K})) = \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_{n-1}(\mathsf{K}))$ and, by inductive hypothesis,

$$\mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_{n-1}(\mathsf{K})) \asymp \mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_{n-1}(\mathsf{H}))$$

hence, by equalities (13), $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) \asymp \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$.

$\boxed{\lambda_n = \mathsf{X} {\to} \mathsf{H} \colon \mathsf{m}}$ The case $\mathsf{X} = \mathsf{K}$ is symmetric to the previous case with $\lambda_n = \mathsf{H} {\to} \mathsf{K} \colon \mathsf{m}$ and therefore omitted. So, assume that X is a participant of S_1; note that by construction $\mathsf{X} \neq \mathsf{H}$ (cf. Definition 2.3). Then $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}), \mathsf{K})$ has the transition $s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{X}\,\mathsf{H}?\mathsf{m}} s(\mathsf{H})$ which corresponds to an input transition $s_{n-1}(\mathsf{H}) \xrightarrow{\mathsf{X}\,\mathsf{H}?\mathsf{m}} r$. We have that ${}_{\mathsf{H}}\rfloor s_{n-1} \in RS[\![\mathsf{S}_1]\!]$ by inductive hypothesis, and so is ${}_{\mathsf{H}}\rfloor s_{n-1} \xrightarrow{\mathsf{X} \to \mathsf{H} \colon \mathsf{m}} {}_{\mathsf{H}}\rfloor s$ since, by definition of left-projection, ${}_{\mathsf{H}}\rfloor s(\mathsf{H}) = r$. The reachability of the right-projection of s immediately follows by inductive hypothesis, since $s\lfloor_{\mathsf{K}} = s_{n-1}\lfloor_{\mathsf{K}}$.

We now show the compatibility condition. By definition, we have that $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) = \mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_{n-1}(\mathsf{H}))$. Moreover, we necessarily have that $s(\mathsf{K}) = s_{n-1}(\mathsf{K})$. Hence by inductive hypothesis,

$$\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) = \mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s_{n-1}(\mathsf{H})) \asymp \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s_{n-1}(\mathsf{K})) = \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$$

The cases $\lambda_n = \mathsf{X} {\to} \mathsf{K} \colon \mathsf{m}$ and $\lambda_n = \mathsf{K} {\to} \mathsf{X} \colon \mathsf{m}$ are similar to the last two cases above. □

C Proofs for Section 5 (Preserving Properties by Composition)

Theorem 5.1 (Deadlock freedom preservation)**.** *Let* S_1 *and* S_2 *be two* (H, K)*-composable and deadlock-free systems. Then the composed system* $\mathsf{S}_1{}^{\mathsf{H} \leftrightarrow \mathsf{K}}\mathsf{S}_2$ *is deadlock-free.*

Given an H-local CFSM M and a participant $\mathsf{K} \in \mathscr{P} \setminus \{\mathsf{H}\}$, call *connecting* a fresh asymmetric sending state of $\mathrm{gw}(M, \mathsf{K})$ whose next outgoing transition does not have K as receiver.

Proof. We show that if the composed system $\mathsf{S}_1{}^{\mathsf{H} \leftrightarrow \mathsf{K}}\mathsf{S}_2$ reaches a deadlock configuration s then at least one of ${}_{\mathsf{H}}\rfloor s$ and $s\lfloor_{\mathsf{K}}$ is a deadlock. Without loss of generality, we assume that the deadlock is the left-projection; the case where the deadlock is the right-projection is similar.

First, we show that if a participant A from S_1 has an enabled transition in s then some participant in S_1 has a transition enabled in ${}_{\mathsf{H}}\rfloor s$. Note that ${}_{\mathsf{H}}\rfloor s$ is reachable in S_1 by Proposition B.3.

If $\mathsf{A} \neq \mathsf{H}$ then any transition of A enabled in s is also enabled in ${}_{\mathsf{H}}\rfloor s$ since ${}_{\mathsf{H}}\rfloor s(\mathsf{A}) = s(\mathsf{A})$ by Definition A.1. If $\mathsf{A} = \mathsf{H}$, then either of the following cases occurs

- H has enabled an input
  Assume that the input is from K. Then by construction ${}_{\mathsf{H}}\rfloor s(\mathsf{H})$ has a τ-transition enabled in $\mathsf{S}_1(\mathsf{H})$. If the input of H is from a participant A of S_1 then by construction ${}_{\mathsf{H}}\rfloor s(\mathsf{H})$ has an input transition enabled in $\mathsf{S}_1(\mathsf{H})$.
- H has enabled an output.
  Assume that the receiver of such output is K. Then $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H})) \asymp \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$ by Proposition B.3. By definition of nof and of gateway, $\mathsf{nof}_{\mathsf{H},\mathsf{K}}(\mathsf{S}_1(\mathsf{H}), s(\mathsf{H}))$ has an input transition enabled from a participant in S_1, hence $\mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$ has a corresponding output transition enable towards a participant in S_2 by compatibility. By construction (Definition A.1), $s\lfloor_{\mathsf{K}}(\mathsf{K}) = \mathsf{nof}_{\mathsf{K},\mathsf{H}}(\mathsf{S}_2(\mathsf{K}), s(\mathsf{K}))$, hence there is a participant, in particular K, willing to take a transition.

If the receiver of the output from H is a participant of S_1 then, by definition of gateway and configuration projection, we get that in H is willing to perform an output from ${}_{\mathsf{H}}\rfloor s(\mathsf{H})$ in S_1.

- H can perform a τ-transition.

  Then, there is a sequence of transitions of the form $s(\mathsf{H}) \xrightarrow{\tau} \xrightarrow{\mathsf{HX!m}}$ in $\mathsf{S}(\mathsf{H})$ with $\mathsf{X} = \mathsf{K}$ or $\mathsf{X} \neq \mathsf{H}$ participant of S_1. If the former case we can reason as in the previous case when H outputs to K. Otherwise, ${}_{\mathsf{H}}\rfloor s(\mathsf{H})$ has an enabled τ-transition in $\mathsf{S}_1(\mathsf{H})$ by Definition A.1.

If s is a deadlock, by definition of deadlock freedom (cf. Definition 2.5), $s \not\to$ but there are participants in S with enabled transitions in s. Under the assumption that $s\lfloor_{\mathsf{K}}$ is deadlock-free, such participants must belong to S_1. By the cases shown above, ${}_{\mathsf{H}}\rfloor s$ enables some participants in S_1; therefore, there is ${}_{\mathsf{H}}\rfloor s \xrightarrow{\lambda}$ because S_1 is deadlock free by hypothesis. It must be that H is involved in all transitions from ${}_{\mathsf{H}}\rfloor s$ of S_1 otherwise $s \xrightarrow{\lambda}$ since for all $\mathsf{X} \in \mathtt{ptp}(\lambda)$ $s(\mathsf{X}) = {}_{\mathsf{H}}\rfloor s(\mathsf{X})$ (by Definition A.1) contrary to our assumption that s is a deadlock. We proceed by case analysis on λ.

$\boxed{\lambda = \mathsf{H} \to \mathsf{X} \colon \mathsf{m}}$ If $\mathsf{X} \neq \mathsf{K}$ then ${}_{\mathsf{H}}\rfloor s(\mathsf{X}) = s(\mathsf{X}) \xrightarrow{\mathsf{HA?m}}$; also, H would be in a connecting state and, by Definition A.1, ${}_{\mathsf{H}}\rfloor s(\mathsf{H}) = s(\mathsf{H}) \xrightarrow{\mathsf{HA!m}}$. Hence $s \xrightarrow{\lambda}$ contrary to the hypothesis that s is a deadlock configuration.

If $\mathsf{X} = \mathsf{K}$ then we have again a contradiction since $s \xrightarrow{\lambda}$ by Proposition B.3.

$\boxed{\lambda = \mathsf{X} \to \mathsf{H} \colon \mathsf{m}}$ If $\mathsf{X} \neq \mathsf{K}$ then ${}_{\mathsf{H}}\rfloor s(\mathsf{X}) = s(\mathsf{X}) \xrightarrow{\mathsf{AH!m}}$ and ${}_{\mathsf{H}}\rfloor s(\mathsf{X}) \xrightarrow{\mathsf{AH?m}} = s(\mathsf{H})$ Hence $s \xrightarrow{\lambda}$ contrary to the hypothesis that s is a deadlock configuration.

If $\mathsf{X} = \mathsf{K}$ then we have again a contradiction since $s \xrightarrow{\lambda}$ by Proposition B.3.

$\boxed{\lambda = \tau}$ If $s(\mathsf{H}) = p''$ is fresh then $\mathrm{gw}(\mathsf{S}_1(\mathsf{H}), \mathsf{K})$ must have a sequence of transitions

$$p \xrightarrow{\mathsf{AH?m}} p' \xrightarrow{\tau} p'' \xrightarrow{\mathsf{HK!m}} r \tag{14}$$

with A participant of S_1 and p' fresh. (Note that it cannot be $s(\mathsf{H}) = p'$ otherwise $s(\mathsf{H}) \xrightarrow{\tau}$ contradicting $s \not\to$.) By Definition A.1, ${}_{\mathsf{H}}\rfloor s(\mathsf{H}) = r$. Hence, by Proposition B.3 we have $s \xrightarrow{\mathsf{H} \to \mathsf{K} \colon \mathsf{m}}$ contradicting $s \not\to$.

If $s(\mathsf{H})$ is not fresh then H has a τ transition enabled at s (because $s(\mathsf{H}) = {}_{\mathsf{H}}\rfloor s(\mathsf{H})$ by Definition A.1) again contradicting $s \not\to$. □

Lemma C.1. *Let* $\mathsf{S} = \mathsf{S}_1 {}^{\mathsf{H} \leftrightarrow \mathsf{K}} \mathsf{S}_2$ *where* S_1 *and* S_2 *are two* (H, K)*-composable systems with* H *and* K *sequential. Given* $s \in \mathscr{R}(\llbracket \mathsf{S} \rrbracket)$*, if*

(1) either ${}_{\mathsf{H}}\rfloor s \xrightarrow{\lambda} s'$ *in* S_1 *and* $\lambda = \tau \implies {}_{\mathsf{H}}\rfloor s(\mathsf{H}) = s'(\mathsf{H})$

(2) or ${}_{\mathsf{H}}\rfloor s \xrightarrow{\tau} \hat{s}$ *involving* H *in* $\mathsf{S}_1(\mathsf{H})$ *and* $\hat{s}$ *reaching a configuration* $\hat{s}'$ *such that* $\hat{s}' \xrightarrow{\lambda} s'$ *in* S_1 *with* $\lambda = \mathsf{H} \to \mathsf{A} \colon \mathsf{m}$

then there is a run $s \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} \hat{s}$ *in* S *such that* $\lambda_n = \lambda$ *and* ${}_{\mathsf{H}}\rfloor \hat{s} = s'$.

The same holds for the right-projection of S.

Proof. We give the proof for each case.

$\boxed{\text{Case (1)}}$ By case analysis on λ noticing that the case $\lambda = \mathsf{H} \to \mathsf{A} \colon \mathsf{m}$ is not possible since ${}_{\mathsf{H}}\rfloor s$ cannot enable output transitions from H by construction (cf. Definition A.1).

$\lambda = \tau$. Then the τ-transition is executed by $\mathsf{A} \neq \mathsf{H}$ in S_1; hence there is a transition ${}_{\mathsf{H}}\rfloor s(\mathsf{A}) \xrightarrow{\tau} q$ in $\mathsf{S}_1(\mathsf{A})$. Observing that $s(\mathsf{A}) = {}_{\mathsf{H}}\rfloor s(\mathsf{A})$ by Definition A.1 we have the thesis since $s \xrightarrow{\tau} s[\mathsf{A} \mapsto q]$.

$\lambda = \mathsf{A}{\to}\mathsf{H}\colon \mathsf{m}$. We have

$$_{\mathsf{H}}\rfloor s(\mathsf{H}) \xrightarrow{\mathsf{A\,H?m}} p \text{ in } \mathsf{S}_1(\mathsf{H}) \qquad \text{and} \qquad _{\mathsf{H}}\rfloor s(\mathsf{A}) \xrightarrow{\mathsf{A\,H!m}} q \text{ in } \mathsf{S}_1(\mathsf{A})$$

moreover, $s(\mathsf{H}) = _{\mathsf{H}}\rfloor s(\mathsf{H})$ and $s(\mathsf{A}) = _{\mathsf{H}}\rfloor s(\mathsf{A})$. Hence,

$$s \xrightarrow{\mathsf{A}\to\mathsf{H}\colon \mathsf{m}} s'' \qquad \text{with} \qquad s''(\mathsf{X}) = \begin{cases} p, & \text{if } \mathsf{X} = \mathsf{H} \\ q, & \text{if } \mathsf{X} = \mathsf{A} \\ s(\mathsf{X}), & \text{otherwise} \end{cases}$$

by Definition 2.3. Therefore $_{\mathsf{H}}\rfloor s'' = s'$.

$\boxed{\text{Case (2)}}$ Note that $\mathsf{S}_1(\mathsf{H})$ necessarily contains the transitions $_{\mathsf{H}}\rfloor s(\mathsf{H}) \xrightarrow{\tau} \hat{s}'(\mathsf{H}) \xrightarrow{\mathsf{H\,A!m}} r$. Then, by construction, in $\mathsf{gw}(\mathsf{S}_1(\mathsf{H}), \mathsf{K})$ we have

$$_{\mathsf{H}}\rfloor s(\mathsf{H}) \xrightarrow{\mathsf{K\,H?m}} p \xrightarrow{\tau} \hat{s}'(\mathsf{H}) \xrightarrow{\mathsf{H\,A!m}} r \tag{15}$$

note that p and $\hat{s}'(\mathsf{H})$ are fresh, that these are the only transitions from $_{\mathsf{H}}\rfloor s(\mathsf{H})$ to r by sequentiality, and that $s(\mathsf{H}) \in \{_{\mathsf{H}}\rfloor s(\mathsf{H}), p, \hat{s}'(\mathsf{H})$ by construction (cf. Definition A.1). If $s(\mathsf{H}) = _{\mathsf{H}}\rfloor s(\mathsf{H})$ then, by Proposition B.3, we have $s \xrightarrow{\mathsf{K}\to\mathsf{H}\colon \mathsf{m}} \xrightarrow{\tau} s''$ with $s''(\mathsf{H}) = \hat{s}'(\mathsf{H})$. Hence $_{\mathsf{H}}\rfloor s'' = _{\mathsf{H}}\rfloor s[\mathsf{H} \mapsto r]$ and therefore $s'' \xrightarrow{\mathsf{H}\to\mathsf{A}\colon \mathsf{m}} s''[\mathsf{H} \mapsto r, \mathsf{A} \mapsto s'(\mathsf{A})] = s'''$ which yields the thesis noticing that $_{\mathsf{H}}\rfloor s''' = s'$. If $s(\mathsf{H}) = p$ then $s \xrightarrow{\tau} s''$ with $s''(\mathsf{H}) = \hat{s}'(\mathsf{H})$; hence the thesis follows as in the previous case. Finally, if $s(\mathsf{H}) = \hat{s}'(\mathsf{H})$ then $s \xrightarrow{\mathsf{H}\to\mathsf{A}\colon \mathsf{m}} s''$ with $s'' = s[\mathsf{H} \mapsto r, \mathsf{A} \mapsto s'(\mathsf{A})]$ and therefore $_{\mathsf{H}}\rfloor s'' = s'$. □

Theorem 5.5 (Lock-freedom preservation)**.** *Let* S_1 *and* S_2 *be two* (H, K)*-composable and lock-free systems with* H *and* K *sequential. Then the composed system* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *is lock-free.*

Proof. By contradiction, let us assume $\mathsf{S} = \mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ not to be lock-free. Then there is a configuration $s \in \mathscr{R}([\![\mathsf{S}]\!])$ and a participant X not involved in any run from s. Without any loss of generality, we can assume $\mathsf{X} \in \mathsf{S}_1$. We have $_{\mathsf{H}}\rfloor s \in \mathscr{R}([\![\mathsf{S}_1]\!])$ by Proposition B.3 and, by lock-freedom of S_1, $_{\mathsf{H}}\rfloor s$ cannot be a lock of S_1 for X. Hence, there exists a run $_{\mathsf{H}}\rfloor s \xrightarrow{\lambda_0} s_0 \cdots s_{n-1} \xrightarrow{\lambda_n} s_n$ of S_1 with X involved in λ_n. We show that this induces a run from s in S involving X by induction on n.

- If $n = 0$, by Lemma C.1 there is a run $s \xrightarrow{\psi} \xrightarrow{\lambda} s'$ such that $_{\mathsf{H}}\rfloor s' = s_0$ with $\lambda_0 = \lambda$.
- If $n > 0$, we assume that the statement holds for all runs with less than n transitions. If X is involved in λ_i with $0 \leq i < n$ then the thesis follows by inductive hypothesis. Let us therefore assume that X is involved in λ_n only. By repeated application of Lemma C.1, there is a run $s \xrightarrow{\psi_1 \cdot \lambda'_1} s'_1 \cdots \xrightarrow{\psi_{n-1} \cdot \lambda'_{n-1}} s'_{n-1} \xrightarrow{\psi_n \cdot \lambda'_n} s'_n$ in S such that $\lambda'_i = \lambda_i$ and $_{\mathsf{H}}\rfloor s'_i = s_i$ for each $1 \leq i \leq n$.

In both cases s reaches a configuration with a run involving X, which contradicts our assumption. □

Theorem 5.6 (Strong lock freedom preservation)**.** *Let* S_1 *and* S_2 *be two* (H, K)*-composable and strongly lock free systems. Then the composed system* $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ *is strongly lock free.*

Proof. By contradiction, let us assume $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ not to be strongly lock free. This means that there are a reachable configuration s, a participant X, and a maximal run ψ of $\mathsf{S}_1{}^{\mathsf{H}\leftrightarrow\mathsf{K}}\mathsf{S}_2$ such that $s \xrightarrow{\psi}$ and X is not involved in any of those transitions. By the first part of the proof of Theorem 5.1, there exist two run $_{\mathsf{H}}\rfloor s \xrightarrow{\psi_1}$ and $s\lfloor_{\mathsf{K}} \xrightarrow{\psi_2}$ of S_1 and S_2 respectively such that X is involve neither in ψ_1 nor in ψ_2.

- In case ψ is infinite, we get that either ψ_1 or ψ_2 is infinite, and hence maximal.
- In case ψ is finite it is possible to use the second part of the proof of Theorem 5.1 to show that either ψ_1 or ψ_2 is maximal,

In both cases we get a contradiction of the hypothesis that S_1 and S_2 are strong lock free. $\square$

Lang-n-Send Extended: Sending Regular Expressions to Monitors [*]

Matteo Cimini
University of Massachusetts Lowell
Lowell, MA, USA
matteo_cimini@uml.edu

In prior work, Cimini has presented LANG-N-SEND, a π-calculus with language definitions.

In this paper, we present an extension of this calculus called LANG-N-SEND$^{\texttt{+m}}$. First, we revise LANG-N-SEND to work with transition system specifications rather than its language specifications. This revision allows the use of negative premises in deduction rules. Next, we extend LANG-N-SEND with monitors and with the ability of sending and receiving regular expressions, which then can be used in the context of larger regular expressions to monitor the executions of programs.

We present a reduction semantics for LANG-N-SEND$^{\texttt{+m}}$, and we offer examples that demonstrate the scenarios that our calculus captures.

1 Introduction

As the field of software language engineering advances [14], it is increasingly easier for programmers to quickly define and deploy their own programming languages. Cimini has presented in [13] a π-calculus called LANG-N-SEND that accommodates "language-oriented" concurrent scenarios. LANG-N-SEND can define languages with a syntax for structural operational semantics (SOS), and use these languages to execute programs. Processes can also send and receive fragments of operational semantics through channels. LANG-N-SEND enables scenarios that are not typical, in which servers provide language fragments to clients. Clients then can add them to their language, and execute programs with the newly built language. For example, [13] uses LANG-N-SEND to model the scenario in which a server provides the fragment of operational semantics that defines the disrupt operator [6]. A process first establishes whether the program that it is about to execute is safety-critical or not. If it is, the process receives the disrupt operator semantics from the server and adds it to Basic Process Algebra (BPA) [9]. The process then uses this augmented BPA to execute the program in the context of the disrupt operator.

The crux of LANG-N-SEND consists of two operators: Program executions $(\mathscr{L}, trace) \texttt{>}_x\, program$ and `isInTrace`, used as follows:

$$(\mathscr{L}, trace) \texttt{>}_x\, program \parallel x(trace).\texttt{isInTrace}(a, trace) \Rightarrow P \; ; \; Q$$

where $\mathscr{L}$ is a language definition and x is a channel. $(\mathscr{L}, trace) \texttt{>}_x\, program$ uses the operational semantics of $\mathscr{L}$ to prove a transition from *program*. The evaluation of *program* proceeds this way one step at a time. Each transition is labelled, and LANG-N-SEND accumulates the execution trace in *trace*. When *program* terminates, the final trace is sent over the channel x. The process on the right of the parallel operator receives the trace, and analyzes it with $\texttt{isInTrace}(a, trace) \Rightarrow P \; ; \; Q$. This process checks that the label a is one of the labels in *trace*, and continues as P in such a case. Otherwise, it continues as Q.

[*]We have addressed a part of our reviewers' suggestions. We will address the rest by the camera-ready deadline in July.

This paper addresses two limitations of LANG-N-SEND.

First Addition: From Higher-order Logic Programs to Transition System Specifications [13] provides a syntax for language definitions $\mathscr{L}$. This syntax has been specifically devised to represent operational semantics. The semantics of $\mathscr{L}$ is based on higher-order logic programming as realized with hereditary Harrop formulae [20]: $\mathscr{L}$ is compiled into a higher-order logic program $\mathscr{P}$, and LANG-N-SEND computes the steps of *program* using the provabilty relation $\models$ of higher-order logic programming [20], i.e., $\mathscr{P} \models (\longrightarrow\ \mathit{label\ program\ program'})$, for some *program*′.

However, higher-order logic programs of [20], and therefore $\models$, do not contemplate the use of negation. This prevents LANG-N-SEND from defining languages with operators that use negative premises, such as the priority operator [7], timed operators [17], and some formulations of the sequential operator.

In this paper, we revise LANG-N-SEND to adopt transition system specifications (TSSs) [10], a well-known and widely used formalism for SOS specifications. TSSs include negative premises with a well-established semantics [16]. This is a consequential addition: It has been shown that negative premises are actually necessary to express some operators such as the priority operator [3]. That is, SOS with negative premises is strictly more expressive than SOS without them.

Second Addition: Online Monitors and Communication of Regular Expressions Monitoring is a runtime verification technique that is based on executing a program and observing its behavior. Its goal is to establish whether such execution satisfies or violates a correctness property (see [11] for a survey on the subject). There are two types of monitoring: *offline* monitoring and *online* monitoring. Offline monitoring executes the program and records its execution trace. The trace is then analyzed after the execution terminates. Conversely, online monitoring performs its analysis alongside the execution of the program. That is, an online monitor acts after each step of the execution, and analyzes the trace that has been generated up to that point.

LANG-N-SEND includes a rudimental form of offline monitoring with `isInTrace`. This operation only checks whether an action has occurred, and is insufficient for most scenarios. Therefore, this paper extends LANG-N-SEND with a more powerful way of analyzing execution traces. Specifically, we augment LANG-N-SEND with regular expressions, and we add the ability of checking whether a trace satisfies or violates a regular expression.

LANG-N-SEND does *not* include online monitoring. In this paper, we extend LANG-N-SEND with this feature. We do so with an extended form for program executions:

$$(\mathscr{T}, \mathit{trace}) \mathtt{>}_x\ \mathit{program}\ \mathtt{with\ monitors}\ m_1\ m_2\ \cdots\ m_n$$

where $\mathscr{T}$ is a TSS and m_1, m_2, ..., and m_n are online monitors. Each m_i carries the regular expression to be checked during the execution of *program*, and a process to be executed in case such regular expression is violated.

LANG-N-SEND is tailored to express dynamic scenarios where language fragments are sent and received, and where servers are instructed to execute programs received from other processes. In this dynamic context, it is natural to allow servers to also receive, from external processes, the regular expressions to monitor. We have therefore extended LANG-N-SEND with the ability of sending and receiving regular expressions through channels.

We call this new calculus LANG-N-SEND$^{\mathtt{+m}}$ (as in "plus monitoring").

Contributions We present a reduction semantics for LANG-N-SEND$^{+\mathtt{m}}$ in Section 4. To demonstrate the type of scenarios that LANG-N-SEND$^{+\mathtt{m}}$ captures, we provide the following examples in Section 5:

- **Negative premise (Example 1).** A server receives the semantics of the parallel operator from another process, which decides whether parallel processes are allowed to spend idle time or whether they must run with maximal progress.
- **Offline monitoring (Example 2).** A server receives programs from clients, executes them to the end, and checks that the programs have used files correctly (open before read/write operations, and close at the end). The server does so by checking that the final trace is accepted by an appropriate regular expression.
- **Online monitoring and sending/receiving of regular expressions (Example 3).** This example refines Example 2. Programs can perform a privileged action on files as long as they respect a correct sequence of actions. This sequence of actions changes every day, and is provided by an external process. The server receives this sequence as a regular expression, and uses it to install an online monitor for the execution of programs.

We believe that LANG-N-SEND$^{+\mathtt{m}}$ provides a suitable formalism to express these and similar scenarios.

The paper is organized as follows. Section 2 provides the definition of transition system specifications from the literature. Section 3 presents the syntax of LANG-N-SEND$^{+\mathtt{m}}$. Section 4 presents a reduction semantics for LANG-N-SEND$^{+\mathtt{m}}$. Section 5 demonstrates our calculus with the examples described above. Section 6 discusses related work, and Section 7 concludes the paper.

2 Preliminaries: Transition System Specifications

We recall the definitions for transition system specifications from [10, 16].

Definition 1 (Signatures and Terms)**.** *A* signature Σ *is a pair* (F, ar) *where* F *is a set of function symbols, and the function* $ar : F \to \mathbb{N}$ *determines the arity of the functions in* F*. Given a signature* $\Sigma = (F, ar)$*,* $T(\Sigma)$ *is the* set of terms of the signature Σ*, and is defined as the minimal set satisfying the following: (We use the symbol* t *for terms).*

- $V \subseteq T(\Sigma)$*, where* V *is a set of variables,*
- *if* $t_1, \ldots, t_n \in T(\Sigma)$*,* $f \in F$*, and* $ar(f) = n$ *then* $f(t_1, \ldots, t_n) \in T(\Sigma)$*.*

We define $\Sigma_\emptyset$ as the *empty signature* with $\Sigma_\emptyset \triangleq (\{\}, \{\})$, that is, both F and ar are empty sets.

Definition 2 (Transition System Specifications (TSS))**.** *A* transition system specification $\mathscr{T}$ *is a triple* (Σ, L, D)*, where* Σ *is a signature,* L *is a set of labels, and* D *is a set of deduction rules. We use the symbol* λ *for labels. Deduction rules are formed with formulae in the way that we describe below. A* positive formula *is of the form* $t \xrightarrow{\lambda} t'$*, and a* negative formula *is of the form* $t \overset{\lambda}{\not\longrightarrow}$*. A* formula f *is either a positive formula or a negative formula.* Deduction rules *are of the form* (H, f)*, where* H *is a set of formulae called* premises *of the rule, and* f *is a positive formula called* conclusion *of the rule. We write a deduction rule* (H, f) *as* $\dfrac{H}{f}$.

The notion of derivability of formulae for TSSs with negative premises is from [15]. As this definition is standard and we do not use any of its machinery, we do not redefine it, but we write $(\Sigma, L, D) \vdash f$ when the formula f is derived from the TSS (Σ, L, D) according to the semantics of [15].

The following definitions from [16] define the componentwise union of two TSSs.

Definition 3 (Union of Signatures)**.** *Given two signatures $\Sigma_1 = (F_1, ar_1)$ and $\Sigma_2 = (F_2, ar_2)$ such that $f \in F_1 \cap F_2 \Rightarrow ar_1(f) = ar_2(f)$, we have*

$$\Sigma_1 \oplus \Sigma_2 = (F_1 \cup F_2, ar'), \textit{ with } ar'(f) = \begin{cases} ar_1(f), f \in F_1 \\ ar_2(f), \textit{otherwise} \end{cases}$$

Definition 4 (Union of TSSs)**.** *Given two TSSs (Σ_1, L_1, D_1) and (Σ_2, L_2, D_2) such that $\Sigma_1 \oplus \Sigma_2$ is defined, we define $(\Sigma_1, L_1, D_1) \oplus (\Sigma_2, L_2, D_2) = (\Sigma_1 \oplus \Sigma_2, L_1 \cup L_2, D_1 \cup D_2)$.*

As an example, we define a TSS for a subset of CCS with inaction *nil*, a unary operator for each action a of a finite set *Act*, and the parallel operator $\|$. (As usual, *Act* also contains complement actions which can be denoted as $\overline{a}$ for any action a.) We call this subset *partialCCS*. The set of variables V of *partialCCS*'s TSS ranges over p, q, and so on. We define *partialCCS* as follows.

$D \triangleq$

$$\{a.p \xrightarrow{a} p, \quad \frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q}, \quad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'},$$

$$\frac{p \xrightarrow{\tau} p'}{p \parallel q \xrightarrow{\tau} p' \parallel q}, \quad \frac{q \xrightarrow{\tau} q'}{p \parallel q \xrightarrow{\tau} p \parallel q'},$$

$$\frac{p \xrightarrow{a} p' \quad q \xrightarrow{\overline{a}} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'}, \quad \frac{p \xrightarrow{\overline{a}} p' \quad q \xrightarrow{a} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'}\}$$

partialCCS $\triangleq ((\{nil, \|\} \cup Act, ar), Act \cup \{\tau\}, D)$, where *ar* assigns the arity 0 to *nil*, the arity 2 to $\|$, and the arity 1 to every element of *Act*.

3 Syntax of Lang-n-Send$^{\texttt{+m}}$

The syntax of Lang-n-Send$^{\texttt{+m}}$ is defined as follows. We assume a set of channels x, y, z, and so on. We assume that this set and the sets Fs, Ls, and Vs of TSSs (see Definition 1 and 2) are pairwise disjoint. (Recall that $\mathscr{T}$ denotes a TSS, and t is a term. We also use the notation $\widetilde{\cdot}$ for finite sequences.)

$$\begin{array}{llll}
\text{Language Builder} & \ell & ::= & \mathscr{T} \mid \ell \ \texttt{union}\ \ell \\
\text{Regular Expression} & \mathscr{E} & ::= & \lambda \mid \varepsilon \mid \mathscr{E}\cdot\mathscr{E} \mid \mathscr{E} \,{\scriptstyle|}\, \mathscr{E} \mid \mathscr{E}^* \\
\text{Trace as Reg. Exp.} & \mathscr{E}_{tr} & ::= & \lambda \mid \varepsilon \mid \mathscr{E}_{tr}\cdot\mathscr{E}_{tr} \\
\text{Transmittable} & e & ::= & x \\
\textit{(language builders)} & & & \mid \mathscr{T} \mid e\ \texttt{union}\ e \\
\textit{(reg. exp.)} & & & \mid \lambda \mid \varepsilon \mid e\cdot e \mid e \,{\scriptstyle|}\, e \mid e^* \\
\textit{(terms)} & & & \mid t \\
\text{Monitor} & m & ::= & \mathscr{E} \Rightarrow P \\
\text{Process} & P,Q,R & ::= & \mathbf{0} \mid x(y).P \mid \overline{x}\langle e\rangle.P \mid P \parallel Q \mid P+Q \mid \nu x.P \mid\ !P \\
\textit{(online monitoring)} & & & \mid \texttt{(}e,\mathscr{E}_{tr}\texttt{)>}_x\ e\ \texttt{with monitors}\ \widetilde{m} \\
\textit{(offline monitoring)} & & & \mid \texttt{verifyThis}(e,e)\ \texttt{?}\ P\ \texttt{:}\ Q \\
\textit{(checking labels)} & & & \mid \texttt{labels}(\widetilde{\lambda},e)\ \texttt{?}\ P\ \texttt{:}\ Q
\end{array}$$

Language builder expressions ℓ evaluate to TSSs $\mathscr{T}$. We can combine two TSSs with `union`, which performs the union operations that we have seen in Section 2.

LANG-N-SEND$^{\texttt{+m}}$ executes programs and keeps track of their execution trace. We analyze these traces with regular expressions over the set of labels L as the alphabet. The grammar of regular expressions $\mathscr{E}$ is standard (with λs as atomic symbols). To recall: ε is the empty string, we use an explicit concatenation operator $\cdot$ (though literature often uses juxtaposition), $|$ is the alternation operator, and $\mathscr{E}^*$ is the Kleene closure of $\mathscr{E}$. As usual, the semantics of a regular expression $\mathscr{E}$ is a set of strings. We denote this set with $[\![\mathscr{E}]\!]$. The definition of $[\![\mathscr{E}]\!]$ is standard and we omit it here [18].

Traces are finite strings of labels $\lambda_1\lambda_2\ldots\lambda_n$. We represent traces with regular expressions of the form $\lambda_1 \cdot \lambda_2 \ldots \cdot \lambda_n$, i.e., a concatenation of labels. Therefore, we have traces $\mathscr{E}_{tr}$ as a special case of regular expressions that denote a singleton set with one string.

LANG-N-SEND$^{\texttt{+m}}$ can send and receive *transmittable expressions e* through channels. Transmittable expressions are channels, language builder expressions, regular expressions, and terms. Similarly to the π-calculus, channel names x, y, and so on, are binding variables for channels. Additionally, LANG-N-SEND$^{\texttt{+m}}$ uses x, y, and so on, as binding variables for language builder expressions, regular expressions, and terms, as well. Transmittable expressions can be language builder expressions that contain variables such as $\mathscr{T}$ `union` x, where x will be substituted after a communication takes place. Similarly, transmittable expressions can be regular expressions that contain variables such as $\mathscr{E} \cdot x$, where x will be substituted later. Notice that LANG-N-SEND$^{\texttt{+m}}$ processes are such that expressions like $\mathscr{T}$ `union` x and $\mathscr{E} \cdot x$ will have x already substituted when we reach the moment where these expressions are used.

LANG-N-SEND$^{\texttt{+m}}$ contains the processes of the π-calculus, except that the output prefix sends transmittable expressions. Furthermore, LANG-N-SEND$^{\texttt{+m}}$ contains the following processes.

$(e_1, \mathscr{E}_{tr}) \texttt{>}_x\, e_2$ `with monitors` $\widetilde{m}$ is a *program execution* with online monitors $\widetilde{m}$. This process evaluates e_1 to a TSS $\mathscr{T}$, and we have that e_2 is a term t at the moment this process is activated. (We offer some remarks on type errors at the end of this section.) The term t is the program to be executed. This process executes the program t according to the semantics of $\mathscr{T}$. To do so, we use the derivability of formulae of TSSs to derive a transition from t. Program executions evaluate t one step at a time. Each of these transitions has a label, and we concatenate these labels in $\mathscr{E}_{tr}$. We assume that every program execution starts with the empty string ε. Therefore, $\mathscr{E}_{tr}$ is the trace of the execution up to a certain point. After each transition, we check that the current trace satisfies all the monitors $\widetilde{m}$. Each monitor m contains a regular expression and a process P. If the regular expression does not validate the trace then the whole program execution is discarded and P is executed instead. If there are multiple monitors that are not satisfied, LANG-N-SEND$^{\texttt{+m}}$ non-deterministically executes the process of one of them. (We purposely under-specify this part. Actual implementations may fix a selection method, for example based on the order in which monitors appear.)

When the execution of t terminates, the trace $\mathscr{E}_{tr}$ is sent over the channel x.

LANG-N-SEND$^{\texttt{+m}}$ accommodates offline monitors, as well, which analyze the trace of the whole execution after t terminates. We do so in the following way. As just described, the trace $\mathscr{E}_{tr}$ can be received over the channel x. Afterwards, it can be used with the process `verifyThis`$(\mathscr{E}_{tr}, \mathscr{E})$ `?` P `:` Q, which behaves as P if the regular expression $\mathscr{E}$ validates the trace $\mathscr{E}_{tr}$, and behaves as Q otherwise. More specifically, the operator `verifyThis` works in a slightly more general form: `verifyThis`(e_1, e_2) `?` P `:` Q, where e_1 and e_2 are regular expressions $\mathscr{E}_1$ and $\mathscr{E}_2$ at the moment this process is activated. Notice that $\mathscr{E}_1$ is not necessarily some $\mathscr{E}_{tr}$. `verifyThis` checks whether $\mathscr{E}_2$ subsumes $\mathscr{E}_1$, i.e., $[\![\mathscr{E}_1]\!] \subseteq [\![\mathscr{E}_2]\!]$ [1]. We offer this general form as a convenience for programmers. For example, a server may already be planning to run an online monitor with $\mathscr{E}_2$, and may receive $\mathscr{E}_1$ from another process with instructions to monitor it, as well. With `verifyThis`$(\mathscr{E}_1, \mathscr{E}_2)$ `?` P `:` Q, the server can program P to run a monitor with $\mathscr{E}_2$ only, as

[1]The inclusion problem is decidable for regular expressions [18, 19].

it subsumes $\mathscr{E}_1$, as in

$$\begin{array}{rl}\texttt{verifyThis}(\mathscr{E}_1,\mathscr{E}_2)\ \texttt{?}\ \texttt{(}\mathscr{T},\varepsilon\texttt{)>}_x\, t\ \texttt{with monitors}\ \mathscr{E}_2 \Rightarrow \overline{\textit{monitor-fail}}.\mathbf{0} & \\ \texttt{:} & \\ \texttt{(}\mathscr{T},\varepsilon\texttt{)>}_x\, t\ \texttt{with monitors}\ \mathscr{E}_1 \Rightarrow \overline{\textit{monitor-fail}}.\mathbf{0} & \\ \mathscr{E}_2 \Rightarrow \overline{\textit{monitor-fail}}.\mathbf{0} & \end{array}$$

Notice that when `verifyThis` is used with a trace as in $\texttt{verifyThis}(\mathscr{E}_{tr},\mathscr{E})\ \texttt{?}\ P\ \texttt{:}\ Q$, then $[\![\mathscr{E}_{tr}]\!]$ is a singleton set with a string and $[\![\mathscr{E}_{tr}]\!] \subseteq [\![\mathscr{E}]\!]$ holds whenever that string is in $[\![\mathscr{E}]\!]$.

A process $\texttt{labels}(\widetilde{\lambda},e)\ \texttt{?}\ P\ \texttt{:}\ Q$, where e evaluates to a TSS $\mathscr{T}$, checks whether the set of labels of $\mathscr{T}$ is a subset of the labels $\widetilde{\lambda}$. In such a case, the process behaves as P. Otherwise, it behaves as Q. As processes may receive TSSs from other processes, this operation is useful to check, before executing programs, that a TSS works with the expected actions.

Some Remarks on Type Errors The syntax of LANG-N-SEND$^{\texttt{+m}}$ does not rule out type errors such as $\mathscr{T}$ `union` $\mathscr{E}$, $\mathscr{E}\cdot\mathscr{T}$, and similar. As future work, we would like to design a type system that rejects these type errors.

4 A Reduction Semantics for LANG-N-SEND$^{\texttt{+m}}$

Figure 1 shows the reduction semantics of LANG-N-SEND$^{\texttt{+m}}$ in two parts. The first part of Figure 1, that is above the horizontal line, contains the standard definition of the structural congruence $\equiv$ of the π-calculus, and includes the reduction rules of the π-calculus that are also part of the semantics of LANG-N-SEND$^{\texttt{+m}}$ [21, 22]. The second part of Figure 1, that is below the horizontal line, contains the rest of the reduction semantics.

The main reduction relation is $\longrightarrow$. As in [13], this relation makes use of two auxiliary relations: $\longrightarrow_{\mathsf{lan}}$ evaluates language builder expressions ℓ into TSSs, and $\longrightarrow_{\mathsf{exe}}$ handles program executions.

Rule (COMM) realizes the communication of transmittable expressions. Substitution $P\{e/y\}$ substitutes the free occurrences of y in P with e. This substitution is capture-avoiding, its definition is straightforward, and therefore we do not show it. Notice that LANG-N-SEND$^{\texttt{+m}}$ adopts a call-by-name style for transmitting language fragments. In particular, an output prefix $\overline{x}\langle\mathscr{T}_1\ \texttt{union}\ \mathscr{T}_2\rangle.P$ transmits the whole expression $\mathscr{T}_1$ `union` $\mathscr{T}_2$ without evaluating it, as it will be evaluated when it is used[2].

Rule (EXEC) handles program executions and simply defers to $\longrightarrow_{\mathsf{exe}}$-transitions. Rule (EXEC-CTX) evaluates ℓ when it is not a TSS yet.

Rule (VERIFY-SUCCESS) checks whether $\mathscr{E}_2$ subsumes $\mathscr{E}_1$ with $[\![\mathscr{E}_1]\!] \subseteq [\![\mathscr{E}_2]\!]$. In that case, the process takes a transition to P. Rule (VERIFY-FAIL) fires whenever $\mathscr{E}_2$ does not subsumes $\mathscr{E}_1$, and executes Q.

Rule (LABELS-SUCCESS) checks whether the labels of the TSS given as second argument are from the set of labels given as first argument. In that case, the process takes a transition to P. Rule (LABELS-FAIL) fires whenever that is not the case, and executes Q. Rule (LABELS-CTX) evaluates ℓ when it is not a TSS yet.

Rule (UNION) performs the union of two TSSs with the operation $\oplus$ defined in Section 2. Rules (UNION-CTX1) and (UNION-CTX2) evaluate the first and second argument of `union`, respectively.

[2]We thank our anonymous reviewers for suggesting a call-by-name approach, which simplifies our calculus. Interested readers can find a call-by-value version of our calculus in Appendix A.

Reduction Semantics $\boxed{P \equiv P,\, P \longrightarrow P,\ \ell \longrightarrow_{\mathsf{lan}} \ell,\ P \longrightarrow_{\mathsf{exe}} P}$

$$P \parallel \mathbf{0} \equiv P \qquad P \parallel Q \equiv Q \parallel P \qquad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$$

$$P + \mathbf{0} \equiv P \qquad P + Q \equiv Q + P \qquad (P + Q) + R \equiv P + (Q + R) \qquad !P \equiv P \parallel !P$$

$$\nu x.\mathbf{0} \equiv \mathbf{0} \qquad \nu x.\nu y.P \equiv \nu y.\nu x.P \qquad \nu x.(P \parallel Q) \equiv \nu x.P \parallel Q,\ \textit{if } x \textit{ is not a free name of } Q$$

$$\frac{P_1 \longrightarrow P_1'}{P_1 + P_2 \longrightarrow P_1'} \qquad \frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2} \qquad \frac{P \longrightarrow P'}{\nu x.P \longrightarrow \nu x.P'} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

(COMM)
$$x(y).P \parallel \overline{x}\langle e\rangle.Q \longrightarrow P\{e/y\} \parallel Q$$

(EXEC)
$$\frac{(\mathscr{T}, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \longrightarrow_{\mathsf{exe}} P}{(\mathscr{T}, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \longrightarrow P}$$

(EXEC-CTX)
$$\frac{\ell \longrightarrow_{\mathsf{lan}} \ell'}{(\ell, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \longrightarrow (\ell', \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m}}$$

(VERIFY-SUCCESS)
$$\frac{[\![\mathscr{E}_1]\!] \subseteq [\![\mathscr{E}_2]\!]}{\texttt{verifyThis}(\mathscr{E}_1, \mathscr{E}_2)\ \texttt{?}\ P\ \texttt{:}\ Q \longrightarrow P}$$

(VERIFY-FAIL)
$$\frac{[\![\mathscr{E}_1]\!] \not\subseteq [\![\mathscr{E}_2]\!]}{\texttt{verifyThis}(\mathscr{E}_1, \mathscr{E}_2)\ \texttt{?}\ P\ \texttt{:}\ Q \longrightarrow Q}$$

(LABELS-SUCCESS)
$$\frac{\mathscr{T} = (\Sigma, L, D) \quad \widetilde{\lambda} = \lambda_1, \cdots, \lambda_n \quad L \subseteq \{\lambda_1, \cdots, \lambda_n\}}{\texttt{labels}(\widetilde{\lambda}, \mathscr{T})\ \texttt{?}\ P\ \texttt{:}\ Q \longrightarrow P}$$

(LABELS-FAIL)
$$\frac{\mathscr{T} = (\Sigma, L, D) \quad \widetilde{\lambda} = \lambda_1, \cdots, \lambda_n \quad L \not\subseteq \{\lambda_1, \cdots, \lambda_n\}}{\texttt{labels}(\widetilde{\lambda}, \mathscr{T})\ \texttt{?}\ P\ \texttt{:}\ Q \longrightarrow Q}$$

(LABELS-CTX)
$$\frac{\ell \longrightarrow_{\mathsf{lan}} \ell'}{\texttt{labels}(\widetilde{\lambda}, \ell)\ \texttt{?}\ P\ \texttt{:}\ Q \longrightarrow \texttt{labels}(\widetilde{\lambda}, \ell')\ \texttt{?}\ P\ \texttt{:}\ Q}$$

(UNION)
$$\mathscr{T}_1\ \texttt{union}\ \mathscr{T}_2 \longrightarrow_{\mathsf{lan}} \mathscr{T}_1 \oplus \mathscr{T}_2$$

(UNION-CTX1)
$$\frac{\ell_1 \longrightarrow_{\mathsf{lan}} \ell_1'}{\ell_1\ \texttt{union}\ \ell_2 \longrightarrow_{\mathsf{lan}} \ell_1'\ \texttt{union}\ \ell_2}$$

(UNION-CTX2)
$$\frac{\ell_2 \longrightarrow_{\mathsf{lan}} \ell_2'}{\ell_1\ \texttt{union}\ \ell_2 \longrightarrow_{\mathsf{lan}} \ell_1\ \texttt{union}\ \ell_2'}$$

(PROGRAM-STEP)
$$\frac{\mathscr{T} \vdash t \xrightarrow{\lambda} t' \quad \widetilde{m} \equiv \mathscr{E}_1 \Rightarrow P_1 \cdots \mathscr{E}_n \Rightarrow P_n \quad [\![\mathscr{E}_{tr} \cdot \lambda]\!] = \{s\} \quad s \in [\![\mathscr{E}_i]\!]\ \textit{for all}\ 1 \leq i \leq n}{(\mathscr{T}, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \longrightarrow_{\mathsf{exe}} (\mathscr{T}, \mathscr{E}_{tr} \cdot \lambda) \texttt{>}_x\, t' \;\texttt{with monitors}\; \widetilde{m}}$$

(MONITOR-FAIL)
$$\frac{\mathscr{T} \vdash t \xrightarrow{\lambda} t' \quad \widetilde{m} \equiv \mathscr{E}_1 \Rightarrow P_1 \cdots \mathscr{E}_n \Rightarrow P_n \quad [\![\mathscr{E}_{tr} \cdot \lambda]\!] = \{s\} \quad s \notin [\![\mathscr{E}_i]\!]\ \textit{for some}\ 1 \leq i \leq n}{(\mathscr{T}, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \longrightarrow_{\mathsf{exe}} P_i}$$

(PROGRAM-END)
$$\frac{\mathscr{T} \not\vdash t \xrightarrow{\lambda} t'}{(\mathscr{T}, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \longrightarrow_{\mathsf{exe}} !\overline{x}\langle \mathscr{E}_{tr}\rangle.\mathbf{0}}$$

Figure 1: Reduction semantics of LANG-N-SEND$^{\texttt{+m}}$.

$$
\begin{aligned}
&\mathit{timeManagementProvider} \triangleq\\
&\qquad !\mathit{whatTask}(y).(\overline{\mathit{getTimeManagement}}\langle \mathit{parallel}\rangle \;+\; \overline{\mathit{getTimeManagement}}\langle \mathit{parallel\text{-}max\text{-}progess}\rangle)\\[1em]
&\mathit{server} \triangleq\\
&\qquad !(\mathit{task}_1(x).\overline{\mathit{whatTask}}\langle \mathit{task}_1\rangle.\mathit{getTimeManagement}(l).\texttt{(}\mathit{almostTPA}\ \texttt{union}\ l,\varepsilon\texttt{)>}_x\ \mathit{tpa\_program}_1\\
&\qquad\ +\\
&\qquad\ \mathit{task}_2(x).\overline{\mathit{whatTask}}\langle \mathit{task}_2\rangle.\mathit{getTimeManagement}(l).\texttt{(}\mathit{almostTPA}\ \texttt{union}\ l,\varepsilon\texttt{)>}_x\ \mathit{tpa\_program}_2)\\
&\mathit{system} \triangleq (\mathit{server} \parallel \mathit{timeManagementProvider} \parallel \mathit{client}_1 \parallel \mathit{client}_2\ \ldots\ \parallel \mathit{client}_n)
\end{aligned}
$$

Figure 2: Server decides idle time vs maximal progress (negative premises).

Rule (PROGRAM-STEP) handles program executions $\texttt{(}\mathscr{T},\mathscr{E}_{tr}\texttt{)>}_x\ t\ \texttt{with monitors}\ \widetilde{m}$. We use the derivability relation $\vdash$ of TSSs to check that a formula $t \xrightarrow{\lambda} t'$ is provable for some t' and some label λ. We then check that all the regular expressions of the monitors $\widetilde{m}$ validate the trace up to that point, which is $\mathscr{E}_{tr}$ with the label λ appended. To do so, we first compute the string s of $\mathscr{E}_{tr}\cdot\lambda$ with $[\![\mathscr{E}_{tr}\cdot\lambda]\!] = \{s\}$. (Recall that the semantics of $\mathscr{E}_{tr}\cdot\lambda$ is a set with one string.) Then we check that s belongs to the semantics of each $\mathscr{E}_i$ of the monitors (with $s \in [\![\mathscr{E}_i]\!]$).

Rule (MONITOR-FAIL) is similar to (PROGRAM-STEP) except that it fires when there exists a regular expression $\mathscr{E}_i$ that does not validate the current trace. In this case the transition takes a step to the corresponding process P_i specified by the failing monitor. Notice that this transition is non-deterministic when there are multiple regular expressions $\mathscr{E}_i$ that fail.

Rule (PROGRAM-END) detects that a step is not provable for t. Then, the execution of t is terminated. We spawn a replicated output prefix that sends the trace over the channel x. The reason for replicating this output is that there may be more than one process that is interested in analyzing the trace, as we shall see in our second example of Section 5.

Deadlocks LANG-N-SEND$^{\texttt{+m}}$ processes can deadlock in as much the same way that π-calculus processes can. Additionally, our processes can deadlock due to the erroneous use of its operators. For example, no reduction rule applies to a process of the form $\texttt{(}\mathscr{T}\ \texttt{union}\ \mathscr{E},\mathscr{E}_{tr}\texttt{)>}_x\ t\ \texttt{with monitors}\ \widetilde{m}$ because $\mathscr{T}\ \texttt{union}\ \mathscr{E}$ is not a valid language builder expression ℓ. Similarly, no reduction rule applies to a process of the form $\texttt{verifyThis(}\mathscr{E}_{tr},\mathscr{E}\cdot\mathscr{T}\texttt{)}\ \texttt{?}\ P\ \texttt{:}\ Q$ because $\mathscr{E}\cdot\mathscr{T}$ is not a valid regular expression $\mathscr{E}$. A type system that rules out these type errors would also rule out this type of deadlocks. (Appendix A shows a version of the calculus with a more refined grammar where these deadlock situations do not occur.)

5 Examples

Example 1 (Negative Premises) Our first example makes use of the newly-added feature to use negative premises in the context of processes that communicate languages. In this example, we have a server that decides whether parallel processes are allowed to spend idle time or whether they must run with maximal progress. We define the TSS of a subset of Hennessy and Regan's Process Algebra for Timed Systems (TPA) [17]. We consider a subset of TPA with inaction *nil*, unary operators $a.P$ for each of the actions a of a finite set *Act*, and the parallel operator $\parallel$. The transitions of TPA are labeled with actions of *Act*, the silent action τ, and the label σ for the passing of idle time. The transition $P \xrightarrow{\sigma} P$ means that the process P spends idle time.

We define *almostTPA* to be the subset of TPA just described. However, we also omit the rule for the passing of idle time in case of the parallel operator. We first define the set of rules D_{tpa^-}.

$$D_{tpa^-} = \{a.P \xrightarrow{\sigma} a.P,\ nil \xrightarrow{\sigma} nil\}.$$

We then define *almostTPA* as an extension of *partialCSS* of Section 2. (Recall that $\Sigma_\emptyset$ is the empty signature defined in Section 2.)

$$almostTPA \triangleq partialCSS \oplus (\Sigma_\emptyset, \{\sigma\}, D_{tpa^-}).$$

We can complete *almostTPA* by including a way to allow time to pass for a parallel operation. For example, we can add either of the following rules.

$$\text{(PAR-IDLE)}\ \frac{p \xrightarrow{\sigma} p' \qquad q \xrightarrow{\sigma} q'}{p \parallel q \xrightarrow{\sigma} p' \parallel q'} \qquad \text{(PAR-MAX)}\ \frac{p \xrightarrow{\sigma} p' \qquad q \xrightarrow{\sigma} q' \qquad p \parallel q \stackrel{\tau}{\not\longrightarrow}}{p \parallel q \xrightarrow{\sigma} p' \parallel q'}$$

(PAR-IDLE) lets the two processes spend idle time, if both processes can. Conversely, (PAR-MAX) implements *maximal progress* and allows idle time to pass only so long that the two processes cannot communicate. (TPA uses (PAR-MAX) in [17].)

We define the two rules in the context of empty TSSs, so that we can conveniently add them to *almostTPA* with our union operator.

$$parallel \triangleq (\Sigma_\emptyset, \{\}, \{\text{(PAR-IDLE)}\})$$
$$parallel\text{-}max\text{-}progess \triangleq (\Sigma_\emptyset, \{\}, \{\text{(PAR-MAX)}\})$$

Figure 2 shows our example. *server* is a server that offers two services, $task_1$ and $task_2$. Upon a request from a client, *server* executes the program $tpa\_program_1$ for $task_1$, and $tpa\_program_2$ for $task_2$. These are programs of our subset of TPA. *server* has limited computational resources, and executing programs in maximal progress mode is computationally expensive. Therefore, *server* communicates with another process called *timeManagementProvider* through the channel *whatTask*, and sends the name of the service that has been requested. *timeManagementProvider* non-deterministically decides whether *server* should use maximal progress or not (perhaps based on the urgency of the task, as well as other factors). *timeManagementProvider* sends *parallel* or *parallel-max-progess* through the channel *getTimeManagement*. In other words, *timeManagementProvider* decides the semantics of the parallel operator, insofar idle time is concerned, that *server* must use. Then, *server* completes *almostTPA* with this fragment of TSS before executing the program.

Example 2 (Offline Monitoring) Figure 3 shows an example with offline monitoring. Here, *server* is a server that manages files. Clients send programs to *server*. Clients also send the TSSs with which *server* must execute these programs. *server* is capable of receiving TSSs and executing programs with them. However, the only actions that *server* supports are the following actions on files: `open`, `read`, `write`, and `close`. In other words, clients can define any TSS they wish, and any SOS operator they wish. Whichever operators they use, however, must compute transitions to open, read, write, and close

$$\begin{aligned}
&\mathit{allowedLabels} \triangleq \texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}\\
&\mathit{fileProtocol} \triangleq \texttt{open} \cdot (\texttt{read} \mid \texttt{write})^* \cdot \texttt{close}\\
&\\
&\mathit{server} \triangleq\ !(\mathit{getProgram}(l,w,id,x).\\
&\qquad\qquad \texttt{labels}(\mathit{allowedLabels},l)\ \texttt{?}\\
&\qquad\qquad\quad (\ (l,\varepsilon)\texttt{>}_x\, w \parallel x(tr).\texttt{verifyThis}(tr,\mathit{fileProtocol}^*)\ \texttt{?}\ \mathbf{0}\ \texttt{:}\ \overline{\mathit{flagClient}}\langle id\rangle\ )\\
&\qquad\qquad\quad \texttt{:}\\
&\qquad\qquad\quad \overline{\mathit{invalid\text{-}language}})\\
&\mathit{onlyOneWrite} \triangleq (\texttt{open} \mid \texttt{read} \mid \texttt{close})^* \cdot \texttt{write} \cdot (\texttt{open} \mid \texttt{read} \mid \texttt{close})^*\\
&\mathit{client}_1 \triangleq \nu x.(\overline{\mathit{getProgram}}\langle tss, tss\_program, id, x\rangle\ \parallel\ x(tr).\texttt{verifyThis}(tr,\mathit{onlyOneWrite})\ \texttt{?}\ P\ \texttt{:}\ \mathbf{0})\\
&\mathit{system} \triangleq (\mathit{server} \parallel \mathit{client}_1 \parallel \mathit{client}_2\ \ldots\ \parallel \mathit{client}_n)
\end{aligned}$$

Figure 3: Server checks for the correct use of files (offline monitoring).

files only, as these are the only actions that *server* recognizes. When the program is finished, the trace is sent on a replicated channel and is available to both server and clients.

Our example models the scenario in which both client and server are running an offline monitor to analyze the trace of an execution. We describe both sides below.

server receives the language l and the program w through the channel *getProgram*[3]. *server* also receives the id of the client (as a channel name), and a channel x where to send the trace of the execution of w once it has finished. After receiving these arguments, *server* checks that the set of labels of l is formed with the allowed labels. If this check fails, the server signals an error through the channel *invalid-language*. Otherwise, the server executes w. As there are no online monitors, we simply write $(l,\varepsilon)\texttt{>}_x\, w$. The server is interested in analyzing the trace of this execution, and so it receives the trace at x and runs an offline monitor with `verifyThis`. The server checks that w has used files correctly, i.e., it has opened a file before reading/writing operations, and it has closed the file afterwards. The correct use of a file is expressed with the regular expression *fileProtocol*. As w may have used files multiple times, the server uses `verifyThis` to check that the trace is accepted by $\mathit{fileProtocol}^*$ (with Kleene star). If this check succeeds then the server ends. Otherwise, the server flags the client as an unreliable programmer using the channel *flagClient*.

One of the clients, client_1, is also interested in analyzing the trace of an execution. client_1 verifies that its program has performed exactly one writing operation. This is expressed with the regular expression *onlyOneWrite*. client_1 sends a TSS *tss* and a program *tss_program* (whose details are irrelevant) to the server. It also sends its id and a private channel x. Then, it receives the trace at x, and runs an offline monitor with `verifyThis` to check that the trace is accepted by *onlyOneWrite*. If this check succeeds then client_1 continues as P. Otherwise, it terminates.

Example 3 (Online Monitoring and Sending/Receiving of Regular Expressions) Figure 4 shows an example of online monitoring in LANG-N-SEND$^{+\mathtt{m}}$, and also illustrates the sending/receiving of regular expressions over channels. This example refines our previous example. Here, *server* additionally admits a privileged action on files, `delete`, which deletes a file. However, programs can perform a `delete`-transition only if they know the "password of the day" provided by the process *passwordMan-*

[3]To shorten our notation, *getProgram* sends and receives multiple arguments in polyadic style, though this is shorthand for a sequence of unary input and output prefixes.

$$
\begin{array}{l}
\textit{allowedLabels} \triangleq \texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}, \texttt{sudo}, 0, 1, 2, \ldots, 9, \texttt{delete} \\
\textit{ordinary} \triangleq \texttt{open} \mid \texttt{read} \mid \texttt{write} \mid \texttt{close} \\
\textit{new} \triangleq \texttt{sudo} \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9 \mid \texttt{delete} \\
\\
\textit{passwordManager} \triangleq\ !(\overline{\textit{getPasswordOfTheDay}}\langle 3 \cdot 4 \cdot 5 \cdot 6 \rangle) \\
\textit{server} \triangleq\ !(\textit{getProgram}(l, w, id). \\
\qquad\qquad \texttt{labels}(\textit{allowedLabels}, l)\ \texttt{?} \\
\qquad\qquad\quad \textit{getPasswordOfTheDay}(e). \\
\qquad\qquad\qquad \nu x.(((l, \varepsilon) \texttt{>}_x\ w\ \texttt{with monitors} \\
\qquad\qquad\qquad\qquad (\textit{new}^* \cdot \textit{fileProtocol} \cdot \textit{new}^*)^* \Rightarrow \overline{\textit{flagClient}}\langle id \rangle) \\
\qquad\qquad\qquad\qquad (\textit{ordinary}^* \cdot (\texttt{sudo} \cdot e \cdot \texttt{delete}) \cdot \textit{ordinary}^*)^* \Rightarrow \overline{\textit{flagClient}}\langle id \rangle) \\
\qquad\qquad\qquad\quad \parallel\ x(tr).\overline{\textit{end}}) \\
\qquad\qquad\quad : \\
\qquad\qquad\quad \overline{\textit{invalid-language}}) \\
\textit{system} \triangleq (\textit{server} \parallel \textit{passwordManager} \parallel \textit{client}_1 \parallel \textit{client}_2\ \ldots\ \parallel \textit{client}_n)
\end{array}
$$

Figure 4: Server receives a regular expression to check valid access to `delete` (online monitoring).

ager. Passwords are numeric. The password of the example in Figure 4 is 3456. Programs must first announce their intention to use the privileged action with a `sudo` action. Then, they must perform the actions that correspond to the digits of the password. In other words, *server* also admits actions 0, 1, 2, ..., 9, where, for example, the action 3 can be interpreted as "sent 3" or "pressed 3". Programs can perform `delete` after having performed this sequence of actions. In our example, the correct sequence of actions for using `delete` is `sudo`, 3, 4, 5, 6, and `delete`, in this order.

server receives the language l, the program w, and the client id. (Clients are not interested about the trace in this example, and so they do not send the channel x of the previous example). The server checks that the set of labels of l is formed with the allowed labels. Then, the server receives a regular expression e through the channel *getPasswordOfTheDay*. This represents the fragment of a trace that corresponds to the correct sequence of actions that enables `delete`. The regular expression so received is substituted in lieu of e, as we shall discuss shortly. At this point, the server creates a private channel x and executes the program w giving x as the channel where to receive the final trace. The server also specifies two online monitors for this program execution. The first monitor performs the check on file operations that we have seen in the previous example, except that the check is performed at each step of the execution. Furthermore, the regular expression of the previous example is slightly modified to take into account the new actions of *server*, which may occur before and after *fileProtocol*.

The second monitor checks that `delete` is used properly. The regular expression of this monitor is $(\textit{ordinary}^* \cdot (\texttt{sudo} \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot \texttt{delete}) \cdot \textit{ordinary}^*)^*$ after e has been substituted. This expression checks that this sequence has been used within the other actions. If this check fails then the client is flagged for knowing the wrong password, or not using the correct protocol.

Notice that $\texttt{sudo} \cdot \texttt{sudo} \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot \texttt{delete}$, as well as other acceptable sequences are invalid. We believe that the example sufficiently demonstrates our approach even though our regular expressions could be more refined.

Finally, *server* can detect that all online monitors succeed throughout the execution of w with the input prefix $x(e).\overline{\textit{end}}$. This process signals successful termination through the channel *end*.

6 Related Work

[13] is a direct related work of LANG-N-SEND^{+m}. We have discussed the differences between this paper and [13] in Section 1 (Introduction). Also, [13] transmits language fragments in call-by-value style while LANG-N-SEND^{+m} does so in call-by-name style. [13] employs a union operation on languages but this operation is not standard, and it has been specifically devised to apply to the syntax for languages of [13]. Instead, we use the standard $\oplus$ operator on TSSs. The examples in this paper showcase the added expressiveness of LANG-N-SEND^{+m} over the prior work done in [13].

There are several works on runtime monitoring (see [11] for a survey). Our paper does *not* offer a new monitoring technique. On the contrary, we have taken an existing approach, i.e., monitoring with regular expressions [8, 12, 25], and have integrated it into a calculus with processes that communicate programs, traces, and languages.

Temporal logics such as LTL, and linear fragments of HML, HML with recursion, and the modal μ-calculus (see [1]) can be used in lieu of regular expressions to state properties on traces. Our first draft of LANG-N-SEND^{+m} had LTL formulae [24] in their finite traces interpretation LTL$_f$ [26] in lieu of regular expressions. However, regular expressions are more expressive than LTL$_f$ [26], and their formalism is more widely known and used, so we simply chose to use that instead. We could use an expressive logic but the goal of this paper is not to use the most powerful logic. Rather, we wanted to demonstrate the type of scenarios that LANG-N-SEND^{+m} enables with a sufficiently expressive formalism that is also easy to read. In this light, we believe that regular expressions may be a suitable choice. As future work, we do plan to integrate more expressive logics in LANG-N-SEND^{+m} and make more sophisticated examples.

[2] provides a general framework for monitoring that is based on operational semantics and that has been implemented in the `detectEr` tool chain [4, 5]. It would be interesting to integrate this framework in LANG-N-SEND^{+m} in future work.

The realm of process calculi is tremendously vast and diverse: Process calculi have been augmented with sophisticated operators, and have been applied to a plethora of domains. We are not aware, however, of process calculi where processes send fragments of TSSs or regular expressions through channels.

7 Conclusion

We have presented an extension of LANG-N-SEND ([13]) called LANG-N-SEND^{+m}. As LANG-N-SEND, our calculus is tailored to model language-oriented scenarios where processes send and receive language fragments. LANG-N-SEND^{+m} also addresses two limitations of [13]. We use TSSs rather than LANG-N-SEND's specification syntax that is based on λ-prolog. This allows LANG-N-SEND^{+m} to define SOS specifications with negative premises. Furthermore, we have added monitoring capabilities based on regular expressions. Processes of LANG-N-SEND^{+m} can also send and receive regular expressions.

We have presented a reduction semantics for LANG-N-SEND^{+m}, and we have provided examples that demonstrate the type of programming scenarios that LANG-N-SEND^{+m} captures.

As future work, we would like to design a type system for LANG-N-SEND^{+m}. We also would like to extend LANG-N-SEND^{+m}. We plan to add more operations on TSSs, such as removing rules, and renaming operators. We observe that the difference between (PAR-IDLE) and (PAR-MAX) is the single premise $P \parallel Q \overset{\tau}{\not\longrightarrow}$. It would be interesting to make LANG-N-SEND^{+m} more fine-grained in its capabilities to communicate fragments of TSSs. We plan to add the ability of sending/receiving premises which then can be added to rules. With such an addition, our first example could simply work with TPA with (PAR-IDLE), and add the premise $P \parallel Q \overset{\tau}{\not\longrightarrow}$ on the fly to make it become (PAR-MAX).

TSSs do not include syntax for binding (and neither does [13]). We plan to integrate the nominal transition systems of Parrow et al. [23] in our calculus, which can accommodate binders in SOS specifications. With such an addition, we would like to make examples with the π-calculus and its variants as TSSs that can be sent/received.

We also would like to investigate a suitable notion of bisimilarity equivalence for LANG-N-SEND$^{+\mathtt{m}}$.

References

[1] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir & Karoliina Lehtinen (2019): *Adventures in monitorability: from branching to linear time and back again*. Proc. ACM Program. Lang. 3(POPL), pp. 52:1–52:29, doi:10.1145/3290365. Available at `https://doi.org/10.1145/3290365`.

[2] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir & Karoliina Lehtinen (2019): *An Operational Guide to Monitorability*. In: Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, pp. 433–453, doi:10.1007/978-3-030-30446-1_23. Available at `https://doi.org/10.1007/978-3-030-30446-1_23`.

[3] Luca Aceto & Anna Ingólfsdóttir (2008): *On the expressibility of priority*. Information Processing Letters 109(1), pp. 83–85, doi:10.1016/j.ipl.2008.09.002. Available at `https://doi.org/10.1016/j.ipl.2008.09.002`.

[4] Duncan Paul Attard, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir & Karoliina Lehtinen (2021): *Better Late Than Never or: Verifying Asynchronous Components at Runtime*. In Kirstin Peters & Tim A. C. Willemse, editors: Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings, Lecture Notes in Computer Science 12719, Springer, pp. 207–225, doi:10.1007/978-3-030-78089-0_14. Available at `https://doi.org/10.1007/978-3-030-78089-0_14`.

[5] Duncan Paul Attard & Adrian Francalanza (2016): *A Monitoring Tool for a Branching-Time Logic*. In Yliès Falcone & César Sánchez, editors: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, Lecture Notes in Computer Science 10012, Springer, pp. 473–481, doi:10.1007/978-3-319-46982-9_31. Available at `https://doi.org/10.1007/978-3-319-46982-9_31`.

[6] Jos C. M. Baeten & Jan A. Bergstra (2000): *Mode transfer in process algebra*. Computing Science Reports 00-01, Technische Universiteit Eindhoven.

[7] Jos C. M. Baeten, Jan A. Bergstra & Jan W. Klop (1986): *Syntax and defining equations for an interrupt mechanism in process algebra*. Fundamenta Informaticae 9(2), pp. 127–167.

[8] Howard Barringer, Allen Goldberg, Klaus Havelund & Koushik Sen (2004): *Rule-Based Runtime Verification*. In Bernhard Steffen & Giorgio Levi, editors: Verification, Model Checking, and Abstract Interpretation, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 44–57.

[9] Jan A. Bergstra & Jan W. Klop (1984): *Process Algebra for Synchronous Communication*. Information and Control 60(1-3), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.

[10] Roland Bol & Jan Friso Groote (1996): *The Meaning of Negative Premises in Transition System Specifications*. J. ACM 43(5), pp. 863–914, doi:10.1145/234752.234756. Available at `https://doi.org/10.1145/234752.234756`.

[11] Ian Cassar, Adrian Francalanza, Luca Aceto & Anna Ingólfsdóttir (2017): *A Survey of Runtime Monitoring Instrumentation Techniques*. In: Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017, pp. 15–28, doi:10.4204/EPTCS.254.2.

[12] Feng Chen & Grigore Roşu (2005): *Java-MOP: A Monitoring Oriented Programming Environment for Java*. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, Springer-Verlag, Berlin, Heidelberg, pp. 546–550, doi:10.1007/978-3-540-31980-1_36. Available at `https://doi.org/10.1007/978-3-540-31980-1_36`.

[13] Matteo Cimini (2022): *Lang-n-Send: Processes That Send Languages*. In: *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES 2022)*, 356, Open Publishing Association, pp. 46–56, doi:10.4204/eptcs.356.5. Available at `https://doi.org/10.4204%2Feptcs.356.5`.

[14] Sebastian Erdweg, Tijs Storm, Markus Vlter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabril D. P. Konat, PedroJ. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin Vlist, Guido H. Wachsmuth & Jimi Woning (2013): *The State of the Art in Language Workbenches*. In Martin Erwig, Richard F. Paige & Eric Wyk, editors: *Software Language Engineering*, *Lecture Notes in Computer Science* 8225, Springer, pp. 197–217, doi:10.1007/978-3-319-02654-1_11.

[15] Rob J. van Glabbeek (2004): *The meaning of negative premises in transition system specifications II*. *Journal of Logical and Algebraic Methods in Programming* 60-61, pp. 229–258, doi:10.1016/j.jlap.2004.03.007. Available at `https://doi.org/10.1016/j.jlap.2004.03.007`.

[16] Jan Friso Groote & Frits Vaandrager (1992): *Structured Operational Semantics and Bisimulation as a Congruence*. *Information and Compututation* 100(2), pp. 202–260, doi:10.1016/0890-5401(92)90013-6. Available at `https://doi.org/10.1016/0890-5401(92)90013-6`.

[17] Matthew C. B. Hennessy & Tim Regan (1995): *A Process Algebra for Timed Systems*. *Information and Computation* 117, pp. 221–239.

[18] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2006): *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] Albert R. Meyer & Larry J. Stockmeyer (1972): *The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space*. In: *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (Swat 1972)*, SWAT '72, IEEE Computer Society, USA, pp. 125–129, doi:10.1109/SWAT.1972.29. Available at `https://doi.org/10.1109/SWAT.1972.29`.

[20] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*, 1st edition. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9781139021326.

[21] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, I*. *Information and Computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[22] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, II*. *Information and Computation* 100(1), pp. 41–77, doi:10.1016/0890-5401(92)90009-5.

[23] Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramunas Gutkovas & Tjark Weber (2015): *Modal Logics for Nominal Transition Systems*. In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pp. 198–211, doi:10.4230/LIPIcs.CONCUR.2015.198. Available at `https://doi.org/10.4230/LIPIcs.CONCUR.2015.198`.

[24] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pp. 46–57, doi:10.1109/SFCS.1977.32. Available at `https://doi.org/10.1109/SFCS.1977.32`.

[25] Koushik Sen & Grigore Rosu (2003): *Generating Optimal Monitors for Extended Regular Expressions*. *Electron. Notes Theor. Comput. Sci.* 89(2), pp. 226–245, doi:10.1016/S1571-0661(04)81051-X. Available at `https://doi.org/10.1016/S1571-0661(04)81051-X`.

[26] Thomas Wilke (1999): *Classifying Discrete Temporal Properties*. In: *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, STACS'99, Springer-Verlag, Berlin, Heidelberg, pp. 32–46.

A Call-by-value Lang-n-Send$^{\texttt{+m}}$

Call-by-value Lang-n-Send$^{\texttt{+m}}$ differs from the calculus of Section 3 in that

- Language builder expressions are evaluated to TSSs before being communicated.
- It makes use of different binding variables for transmittable values: We assume sets of variables LANG-VAR, REXP-VAR, and TERM-VAR. We have that variables $l \in$ LANG-VAR bind TSSs, variables $e \in$ REXP-VAR bind regular expressions, and variables $w \in$ TERM-VAR bind terms. We assume that all these sets, together with the sets Fs, Ls, and Vs of TSSs, are pairwise disjoint.

The syntax of call-by-value Lang-n-Send$^{\texttt{+m}}$ is the following.

$$
\begin{array}{llcl}
\textsf{Language Builder} & \ell & ::= & l \mid \mathscr{T} \mid \ell \;\texttt{union}\; \ell \\
\textsf{Regular Expression} & \mathscr{E} & ::= & e \mid \lambda \mid \varepsilon \mid \mathscr{E} \cdot \mathscr{E} \mid \mathscr{E} \shortmid \mathscr{E} \mid \mathscr{E}^* \\
\textsf{Trace as Reg. Exp.} & \mathscr{E}_{tr} & ::= & \lambda \mid \varepsilon \mid \mathscr{E}_{tr} \cdot \mathscr{E}_{tr} \\
\textsf{Monitor} & m & ::= & \mathscr{E} \Rightarrow P \\
\textsf{Process} & P,Q,R & ::= & \mathbf{0} \mid x(y).P \mid \overline{x}\langle y \rangle.P \mid P \parallel Q \mid P+Q \mid \nu x.P \mid\ !P \\
\textit{(online monitoring)} & & & \mid (\ell, \mathscr{E}_{tr}) \texttt{>}_x\, t \;\texttt{with monitors}\; \widetilde{m} \\
\textit{(offline monitoring)} & & & \mid \texttt{verifyThis}(\mathscr{E}, \mathscr{E}) \;\texttt{?}\; P \;\texttt{:}\; Q \\
\textit{(communicating languages)} & & & \mid\ x(l).P \mid \overline{x}\langle \ell \rangle.P \\
\textit{(communicating terms)} & & & \mid\ x(w).P \mid \overline{x}\langle t \rangle.P \\
\textit{(communicating reg. exp.)} & & & \mid\ x(e).P \mid \overline{x}\langle \mathscr{E} \rangle.P \\
\textit{(checking labels)} & & & \mid \texttt{labels}(\widetilde{\lambda}, \ell) \;\texttt{?}\; P \;\texttt{:}\; Q
\end{array}
$$

The reduction semantics of call-by-value Lang-n-Send$^{\texttt{+m}}$ is that of Figure 1 except that:

- Rule (COMM) is the standard π-calculus rule for communicating channels.

$$
\begin{array}{l}
\text{(COMM)} \\
x(y).P \parallel \overline{x}\langle z \rangle.Q \longrightarrow P\{z/y\} \parallel Q
\end{array}
$$

- It includes the communication rules for TSSs, regular expressions, and terms.

$$
\begin{array}{c}
\text{(COMM-LANG)} \\
\dfrac{\ell \longrightarrow^*_{\mathsf{lan}} \mathscr{T}}{x(l).P \parallel \overline{x}\langle \ell \rangle.Q \longrightarrow P\{\mathscr{T}/l\} \parallel Q}
\end{array}
$$

$$
\begin{array}{ll}
\text{(COMM-TERM)} & \text{(COMM-REGEXP)} \\
x(w).P \parallel \overline{x}\langle t \rangle.Q \longrightarrow P\{t/w\} \parallel Q & x(e).P \parallel \overline{x}\langle \mathscr{E} \rangle.Q \longrightarrow P\{\mathscr{E}/e\} \parallel Q
\end{array}
$$

Notice that (COMM-LANG) evaluates the language builder expression ℓ. Substitution $P\{\mathscr{T}/l\}$ substitutes the free occurrences of l in P with $\mathscr{T}$. Substitution $P\{\mathscr{E}/e\}$ substitutes the free occurrences of e in P with $\mathscr{E}$. Substitution $P\{t/w\}$ substitutes the free occurrences of w in P with t. Substitutions $P\{\mathscr{T}/l\}$, $P\{t/w\}$, and $P\{\mathscr{E}/e\}$ are capture-avoiding, their definition is straightforward, and therefore we do not show it.

As the grammar above is more refined than that of Section 3, the deadlock situations described at the end of Section 4, such as those for processes $(\mathscr{T}\ \texttt{union}\ \mathscr{E}, \mathscr{E}_{tr}) \texttt{>}_x\, t$ $\texttt{with monitors}$ $\widetilde{m}$ and $\texttt{verifyThis}(\mathscr{E}_{tr}, \mathscr{E} \cdot \mathscr{T})$ $\texttt{?}\ P\ \texttt{:}\ Q$, do not occur. Indeed, these processes are not part of the syntax.

The NiRvAna Project: Noninterference and Reversibility Analysis in Private Blockchains

Marco Bernardo
Dipartimento Scienze Pure e Applicate, Università di Urbino
marco.bernardo@uniurb.it

Claudio Antares Mezzina
Dipartimento Scienze Pure e Applicate, Università di Urbino
claudio.mezzina@uniurb.it

NiRvAna is a three-years research project started in June 2022 and funded by the Italian Ministry for University and Research, whose aim is to study noninterference and reversibility in private blockchains. The project proposal originated from a simple and provoking question: are there situations in which a blockchain transaction should be reversed? The question along with its answer undermine the very bases of blockchain technology: immutability.

1 Context of the NiRvAna Project

Distributed computing has by now become a pervasive technology due to the widespread adoption of electronic devices connected by the Internet infrastructure, which are used by individuals, companies, and institutions to perform an increasing number of daily activities in a digital mode. One of the most prominent examples over the last decade is blockchain technology. It results in a distributed ledger that permanently records transactions taking place among untrusted parties in a decentralized and disintermediated environment, which was devised after the global financial crisis of 2008 to avoid the double spending problem in virtual currency platforms [7].

A blockchain is an append-only ledger that collects transactions. It is composed of blocks linked together through cryptography, each one containing the hash value of the previous block, a timestamp, and transaction data. Once stored, a data block cannot be altered or removed from the blockchain without compromising the validity of all the subsequent blocks. This immutability property certifies that transaction data residing in the blockchain are tamper-proof, thus guaranteeing that everyone can trust the blockchain. The parties form a peer-to-peer communication network adhering to a consensus protocol for block validation. Usually a blockchain is public, meaning that the ledger is accessible by anyone without specific read, write, or validate permissions, with the users being free to enter, leave, and join again the network at any time. Transaction validation is accomplished algorithmically, with no central authority, through a computationally expensive mechanism that discourages potential attackers.

A number of shortcomings affect public, permissionless blockchains, such as the excessive energy consumption required by the consensus protocol and conflicts between data immutability and regulations (right to be forgotten and transaction invalidation due to digital identity theft or contract nullity or infringement, just to mention a few). In the specific case of innovative payment methods, there are also risks of losing monetary sovereignty and undermining financial stability, as witnessed by the fact that many central banks are exploring the issuance of what is called central bank digital currency (CBDC) [4],

Submitted to:
ICE 2022

which would also reduce the costs associated with managing physical cash, promote financial inclusion, and hopefully discourage tax evasion, money laundering, and other illegal activities. As a consequence private, permissioned blockchains are getting momentum, as they could ultimately give businesses a greater degree of control. The reason is that they can be accessed only by authorized people and are governed by a designated person, enterprise, or authority. Methods for reversing illegal transactions are thus needed in a private blockchain, as well as unauthorized information flows from its governance to the various parties must be avoided.

A pillar of blockchain technology is data immutability. Once stored, data cannot be altered or removed from the blockchain. This property certifies that data residing in a public blockchain are tamper-proof, thus creating a digital environment trusted by all parties despite the absence of a central authority certifying user identity and transaction validity. Unfortunately, data immutability may conflict with regulations and the only way out is allowing for some degree of mutability [8, 5]. One example is the right to be forgotten, introduced within the EU in 2016 by the General Data Protection Regulation (GDPR) after the adoption in 2014 of the Regulation on Electronic Identification and Trust Services for Electronic Transactions in the Internal Market (eIDAS). In all these cases, the effects of the considered illegal transactions have to be removed from the blockchain. Again, this seems to be more feasible in a private blockchain rather than in a public one, as in the former it is likely that the owner is endowed with mechanisms for deeming a transaction as reversible due to the permission-based accessibility of the private blockchain. After all, the idea of reversibility has already been considered for virtual currencies, leading for instance to Reversecoin [2], where vault accounts are additionally available with a configurable timeout such that transactions can be reversed before the timeout expires while remaining visible in the ledger, and reversible Initial Coin Offerings [3], based on smart contracts mimicking fund raising in the real world. Also, the idea to retract transactions made by mistake, or made because of identity theft, is becoming utterly evident as witnessed by the proliferation of tools such as Kirobo [1], which provides an undo feature that eliminates the risk of fund losses due to human error.

Developing complex distributed systems like private blockchains is extremely challenging in terms of guaranteeing high levels of proper functioning, data protection, and quality of service. It even becomes a critical issue in CBDC platforms, where errors, data breaches, and poor performance may have economical and social consequences hard to estimate. This calls for a model-based approach in the early design stages so as to enable system property prediction.

2 Approach of the NiRvAna Project

The NiRvAna project (`http://www.sti.uniurb.it/nirvana/`) is about the use of formal methods for the compositional modeling of functional and non-functional aspects of the behavior and the structure of private blockchains. On the analysis side, the focus will be on relevant properties such as noninterference and reversibility. The former is concerned with the absence of information leakage, due to qualitative or quantitative covert channels, from the private blockchain governance to permissioned users. The latter deals with undoing transactions, because of regulation compliance, in a way that timely brings the system back to a previous consistent state. This will be accomplished by developing or extending modeling languages, analysis techniques, and software tools according to a holistic view of trustworthiness that encompasses safety, security, integrity, efficiency, availability, resilience, and ease of use.

We plan to develop a reversible Markovian process algebra equipped with stochastic noninterference techniques that we will apply to the study of private blockchains. We will start by developing models of various aspects of blockchains in general, such as distributed ledgers, consensus protocols, and peer-to-peer asynchronous networks. Then we will focus on private blockchains, in which noninterference and reversibility analyses play a role due to the presence of users with different permissions, and hence different security levels, as well as the necessity of reversing transactions in certain situations, for regulation compliance. Special attention will be paid to CBDC, for which we are already consulting the relevant literature including working papers and reports of the International Monetary Fund, the World Economic Forum, and central banks.

We plan to develop a complete compositional model of a private blockchain written in a reversible variant of PEPA [6], whose correctness, security, and performance properties will be analyzed with our extension of the PEPA Eclipse plug-in. We will then work together with BAX, a firm located in the province of Pesaro and Urbino operating in the field of information and communication technology, to implement a prototype of our verified model of private blockchain. BAX is involved in the regional project MIRACLE (Marche Innovation and Research fAcilities for Connected and sustainable Living Environments) funded by Regione Marche and this will allow us to exploit the computing facilities of a dedicated server farm. We also plan to recruit several PhD and postdoc positions working on the project.

References

[1] *Kirobo: The Safety Net for Defi.* `https://www.kirobo.io/`.

[2] *Reversecoin - The World's First Cryptocurrency with Reversible Transactions.* `https://bitcoinist.com/reversecoin-worlds-first-cryptocurrency-reversible-transactions/`.

[3] *rICO - The Reversible ICO;.* `https://medium.com/lukso/rico-the-reversible-ico-5392bf64318b`.

[4] Michael D. Bordo & Andrew T. Levin (2017): *Central Bank Digital Currency and the Future of Monetary Policy.* Monetary Policy and Payments 3, pp. 143–178.

[5] Rosa M. Garcia-Teruel (2020): *Legal Challenges and Opportunities of Blockchain Technology.* Journal of Property, Planning and Environmental Law 12(2), pp. 129–145.

[6] Jane Hillston (1996): *A Compositional Approach to Performance Modelling.* Cambridge University Press.

[7] Satoshi Nakamoto (2009): *Bitcoin: A Peer-to-Peer Electronic Cash System.* Available at `http://www.bitcoin.org/bitcoin.pdf`.

[8] Eugenia Politou, Fran Casino, Efthimios Alepis & Constantinos Patsakis (2021): *Blockchain Mutability: Challenges and Proposed Solutions.* IEEE Transactions on Emerging Topics in Computing 9(4), pp. 1972–1986.

A Framework for Modeling Behaviour of Aggregated Agents

Michele Loreti
School of Science and Technology,
University of Camerino
Via Madonna delle Carceri, 9,
Camerino (MC), Italy
michele.loreti@unicam.it

Michela Quadrini
School of Science and Technology,
University of Camerino
Via Madonna delle Carceri, 9,
Camerino (MC), Italy
michela.quadrini@unicam.it

Collective-adaptive systems (CASs) consist of many interacting components or agents, which are characterized by attributes, properties, objectives, and functionalities. They compete and cooperate in pursuing individual or collective goals: each agent exhibits individual behaviours to pursue its aims and interacts with the others to reach collective goals. These interactions may lead to unexpected behaviours. Indeed, the aim of an agent my be different from and potentially conflicting with the others. Moreover, agents may be able to adapt at runtime to the changes that can be exerienced in the operating environments.

To cope with these intricacies, methods and tools are needed to forecast and to verify behaviour of CAS. From a specification point of view, one of the main challenges to address is the selection of the formalism used to describe the behaviour of collective systems. Several languages, mainly based on *process algebras*, have been proposed in the literature [4]. Among the others, we can mention here SCEL [5], ABC [1, 2], and CARMA [3]. SCEL is a language that relies on the autonomic components concept (i.e., the collective members) and autonomic-component ensembles, representing collectives [5]. Inspired by SCEL, ABC models the dynamic formation of interaction groups by considering the properties and status of individual members [1, 2]. Finally, CARMA extends attribute based interactions with quantitative information that permits reasoning about performance aspects of CAS . All these approaches are based on the attribute-based paradigm. Agents use attributes for dynamically organizing themselves into ensembles and select partners for interaction. This means that structure of the agent interactions is determined by considering an *agent perspective* that is useful when one is interested in modelling *opportunistic interactions*. However, this is not always the case. Indeed, often the behaviour of an agent is predetermined, while its interaction capabilities depend on the environment where *it is located*. Under this point of view, agents are structured in *groups* that affect the system structure and, consequently, influence both local behaviours, i.e., the one performed by each agent, and the global ones, related to the experience of the whole system.

In this talk, we present a framework that permits modelling the behaviour of agents, which are aggregated in (possibly overlapping) groups. This aggregation depends on the features exposed by the agents and can evolve dynamically. The behaviour of a single agent is defined in terms of the actions it can perform. At each computational step, agents select the action to execute according to a probability distribution that depends on both the agent state and on the composition of groups in the system.

A simple example will be used to motivate the approach and to demonstrate the use of the proposed methodology. We will consider a set of sensors (agents) operating in a given area that must reach a consensus (determined by the majority) about a possible dangerous situation, such as fire or pollution. The agents operate in a fully distributed way without any centralised control. We will show how the proposed methodology permits evaluating the impact of agent distribution over the area on the time needed to reach a consensus.

Submitted to:
ICE 2022

References

[1] Yehia Abd Alrahman, Rocco De Nicola & Michele Loreti (2019): *A calculus for collective-adaptive systems and its behavioural theory*. Information and Computation 268, p. 104457.

[2] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi & Roberto Vigo (2015): *A calculus for attribute-based communication*. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1840–1845.

[3] Luca Bortolussi, Rocco De Nicola, Vashti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti & Mieke Massink (2015): *CARMA: collective adaptive resource-sharing Markovian agents*. arXiv preprint arXiv:1509.08560.

[4] Rocco De Nicola, Gianluigi Ferrari, Rosario Pugliese & Francesco Tiezzi (2020): *A formal approach to the engineering of domain-specific distributed systems*. Journal of Logical and Algebraic Methods in Programming 111, p. 100511.

[5] Rocco De Nicola, Michele Loreti, Rosario Pugliese & Francesco Tiezzi (2014): *A formal approach to autonomic systems programming: the SCEL language*. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 9(2), pp. 1–29.