The Stream-based Service-Centered Calculus: a Foundation for Service-Oriented Programming

Luís Cruz-Filipe
Escola Superior Náutica Infante Dom Henrique, Portugal

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

Francisco Martins
Universidade de Lisboa, Faculdade de Ciências & LaSIGE, Portugal

António Ravara
Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia & CITI, Portugal

Vasco Thudichum Vasconcelos
Universidade de Lisboa, Faculdade de Ciências & LaSIGE, Portugal

Abstract

We give a formal account of SSCC, a calculus for modeling service-based systems, suitable to describe both service composition (orchestration) and the protocols that services follow when invoked (conversation). The calculus includes primitives for defining and invoking services, for isolating conversations (called sessions) among clients and servers, and for orchestrating services. The calculus is equipped with a reduction and a labeled transition semantics related by an equivalence result.

SSCC provides a good trade-off between expressive power for modeling and simplicity for analysis. We assess the expressive power by modeling van der Aalst workflow patterns and an automotive case study from the European project Sensoria. For analysis, we present a simple type system ensuring compatibility of client and service protocols. We also study the behavioral theory of the calculus, highlighting some axioms that capture the behavior of the different primitives.

As a final application of the theory, we define and prove correct some program transformations. These allow to start modeling a system from a typical UML Sequence Diagram, and then transform the specification to match the service-oriented programming style, thus simplifying its implementation using web services technology.

Contents

1 Introduction

Enterprise application integration, either to reuse legacy code, or to combine third-party software modules, has long been tackled by several middleware proposals, namely using message brokers or workflow management systems. As the popularity of using the Web increased, traditional middleware was forced to provide integration across companies. The technologies developed lay in the concept of *Web service*: a way of exposing (to the Web) the functionalities performed by internal systems and making them dynamically searchable (discoverable and accessible through the Web) and composable, allowing adaptability and reusability [1].

1.1 A formal approach to service-oriented programming

Nowadays, Web Services are one of the main technologies used to deploy applications that coordinate behaviors over the Web. Contributing to the success of the technology are the currently available standards [2,6,18,24] that, in order to achieve business goals, allow easy orchestration of different services (distributed and belonging to different organizations), while maximizing interoperability. While standards and programming tools are continuously improving, the formal bases of Service-Oriented Computing (SOC) are still uncertain: there is an urgent need for models and techniques allowing the development of applications in a safe manner, while checking that systems provide the required functionalities. These techniques should be able to deal with the different aspects of services, including their dynamic behavior.

To model systems, and, in particular, to be able to reason and ensure properties about specifications of service-oriented systems, one needs mathematical tools. Process calculi are one suitable tool, providing not only a description language, but a rigorous semantics as well, allowing the proof of relevant properties. Process calculi give precise semantics to system specifications, and they come equipped with a rich toolbox of analysis techniques, including type systems and contextual equivalences. When defining a calculus for SOC, different aspects influence the choice of primitives and of their behavior, and a careful trade-off between expressiveness and suitability to analysis should be found. Our main concerns have been threefold.

Expressiveness of the language: the calculus should be able to express in a direct way the different kinds of interactions that characterize SOC: invocations of services, client-server conversations, and interactions among different client-server pairs. We use three different classes of operators to this end: services, sessions, and streams. We show, via examples, that these are enough to model various kinds of SOC scenarios. We stress in particular the importance of interactions using streams, which is the heart of SSCC orchestration. Other constructs such as tuple spaces or shared memory would be as expressive as streams, but would be more difficult to analyze. Furthermore, primitives similar to our streams are commonly used in practice, for instance when programming Service Oriented Architectures.

Expressiveness of the analysis: the elements to be analyzed should correspond to explicit elements in the calculus. Concerning the three classes of operators in the previous paragraph, service definition is fundamental to speak about service availability. It also allows easy extensions for service discovery based on quality of service. Sessions, instead, allow to analyze client-server compatibility and to study behavioral-based service discovery. Other mechanisms, such as BPEL correlation sets [2], would make these analyses more complex, since they rely on run-time values for determining the communication patterns, spreading the protocol code throughout the whole program. Streams are, on the one hand, used for modeling service coordination, and, on the other hand, needed to study global properties of systems such as deadlock-freedom and the fact that the system satisfies a more abstract specification.

Computability of the analysis: static analysis should be decidable, possibly also efficient to compute. Thus, the allowed communication patterns should be constrained whenever this does not destroy expressiveness. In our calculus, streams and sessions are static, and the dynamism is concentrated in service invocation. To stress the effect of these considerations on the design decisions, we give some "proof of concept" analyses to illustrate how to exploit the features of the calculus.

1.2 Our proposal

We present SSCC (Stream-based, Service-Centered Calculus), a calculus for modeling service-based systems, inspired by SCC [8] and Orc [28, 35], and developed with the above considerations in mind. A prominent feature is that it captures in a direct way the main activities in service-oriented computations: *definition and invocation of services*, *long-running interactions* between the service invoker and the service provider, and *orchestration of complex computations* exploiting services as building blocks.

SSCC builds on SCC [8], and tries to improve, in particular, the suitability for modeling orchestration. To this end, it introduces a new construct, called *stream*, with the aim of collecting the results from some ongoing computations and make them available for new ones. This is the main aspect that differentiates SSCC from CaSPiS [9, 10], the most direct evolution of SCC. While proposing interesting concepts, like sessions, and featuring services as first class entities, SCC looks not fully adequate for service composition. In fact, the only way for a session to interact with other client-server pairs is the **return** primitive, and the functional style of invocation is not adequate for modeling complex patterns of interactions, such as van der Aalst workflow patterns [43]. To overcome these problems we introduce streams and we allow non-persistent service invocations, thus enhancing the expressiveness of the calculus, while making it easier to program with. This design choice has been taken in order to simplify static analysis techniques, trying to find a suitable trade-off between expressiveness of the chosen primitives and suitability to analysis. In particular, since stream names cannot be communicated, their scope is known statically.

Another source of inspiration was Orc [35], a basic programming model for orchestration of Web services. Here a few coordination constructs are used to model the most common patterns, and a satisfying expressiveness is claimed by presenting a formalization of all van der Aalst workflow patterns [19]. However, in order to model the more challenging patterns, special sites (the basic computation entity in Orc) are required, acting, *e.g.*, as semaphores. This is a coordination concern, and in our opinion should be addressed within the language. Thus we introduced more basic mechanisms to tackle all the coordination concerns inside the calculus (most of Orc operators can be expressed as macros in our model). Also, we introduced conversations, which are absent in Orc, to model service behavior (Orc leaves this unspecified).

1.3 Contributions

The main contribution of this paper is SSCC, a language for modeling service-oriented systems based on the session communication paradigm and equipped with a formal semantics. This paper aggregates two conference papers [21, 31] and two technical reports [20, 32]. The preliminary results in the conference papers are extended and fully proved. Thus, much of the material here presented is new.

A clear separation of concerns: conversation and orchestration. As discussed above, the main concerns leading to this language were expressiveness and intuitiveness for modeling and suitability for analysis. Concerning the modeling part, we want to emphasize here some aspects. The calculus allows for the description of service interaction and of service orchestration using distinct mechanisms; the conversation between parties engaged in a service interaction is described by a series of value send/receive, isolated inside a session, while the orchestration of services is performed using the stream operations. The two communication methods are orthogonal, as witnessed for instance by the fact that their scopes give rise to separate nesting hierarchies. This choice simplifies both the type system and the observational semantics, as discussed below. Notice that Orc [35] lacks the conversation primitives, and that both Orc and SCC [8] feature insufficient orchestration. Furthermore, in both SCC [8] and CaSPiS [9, 10] orchestration is not orthogonal w.r.t. sessions, thus making it difficult to program cross-sessions orchestration patterns.

A flexible programming style. Service orchestration and service conversation are both easily structured in SSCC. (Keep in mind that SSCC is a process calculus, not a full-fledged programming language.) We were in fact able to encode all van der Aalst workflow patterns [43] (apart from the ones that require termination), in intelligible code. We tried the same exercise using SCC and the results for some patterns were not satisfying at all. As a more challenging test, we put SSCC at work by modeling the Automotive case study [4] of the European project Sensoria [40]. Again, SSCC passed the test. Actually, it was able to model different programming styles, from object-oriented to session-oriented to request-response oriented. Concerning the analysis part, we concentrated on a type system and on proofs of equivalence based on a contextual equivalence.

A type discipline. We provide a simple static analysis system to check the compatibility between service definition and service invocation, as well as protocol sequentiality. Our type system is inspired in the results achieved for session types [22, 25, 42, 46]. Interestingly, it exploits the separation of concerns between conversation and orchestration. In fact, the type of a process has two components, one concerning its conversation behavior and another concerning its orchestration behavior. This allows to simplify the reasoning in some cases: for instance, the simplified type system for session sequentiality has no need to consider the orchestration part. The same simplification is not possible in other calculi where the two aspects are intertwined.

A coinductive contextual congruence. The contextual congruence in SSCC coincides with a bisimilarity that is also a (non-input) congruence and gives rise to some axiomatic laws. These laws not only clarify the relationship among language constructs, but they also allow for meaningful program transformations.

Program transformations. We exploited the theory for defining and proving correct some program transformations. Those transformations allow to start designing a system from a standard UML Sequence Diagram [3] (based on the object-oriented paradigm). Such a diagram has a direct translation into SSCC, and it induces an object-oriented style. This style does not match the session-oriented style normally used for service-oriented systems. Our first program transformation allows to transform it to fit this second style. However, most implementation technologies based on the service-oriented paradigm do not fully support general sessions, but only request-response communications, which can be seen as a particular case. Our second transformation allows to transform the program to fit this implementation requirement. Interestingly, this transformation exploits the type system for session sequentiality, since it is correct only for sequential sessions.

1.4 Related work

Among process calculi, the π-calculus (and its variants) has been frequently used in SOC. We claim that general purpose concurrent calculi are not suitable for our aims, since the different communication patterns are mixed, and most of the interesting properties are not directly reflected in text of programs. Thus, these calculi do not satisfy the requirements above. Different proposals use types, *e.g.*, session types [22, 25, 42, 46], to solve this problem, but since they allow free π-calculus communications the analysis becomes difficult. We consider our proposal as some kind of tamed π-calculus, with a good trade-off between expressiveness for SOC systems and suitability to analyze SOC-related properties.

This approach was born inside the EU project Sensoria [40] on Software Engineering for Service-Oriented Overlay Computers. The first outcome of this line of research has been SCC [8]. SCC had the merit of introducing services and sessions as first-class entities in the calculus, but lacked good primitives for orchestrating groups of services. The only orchestration-oriented mechanism was the **return** primitive, allowing a session to return a value to its enclosing session, but implementing complex interaction patterns using it was an hard task. We show examples supporting this claim in Section 3.1 and in Appendix B, when modeling workflow patterns [43].

SSCC has been the first proposal [31] extending SCC to provide easier orchestration, followed by the Conversation Calculus [45] and CasPiS [9, 10]. CasPiS approach is the closest to ours. The main difference is that CasPiS uses pipelines for orchestration instead of our streams. Pipelines allow to redirect session communications to another session. The difficulty in using pipelines is that the same messages can be used both for session communication and for orchestration. Instead we use two orthogonal features. The separation of concerns is one of the cornerstones of SSCC approach, since it allows one to use different analysis techniques on messages used for different purposes. For instance, in the type system in Section 2.4, session communications are modeled using behavioral types, while streams have simpler static types. A similar approach would not be as effective in CasPiS. See, e.g., [12] for a type system for CasPiS.

The Conversation Calculus introduces the notion of conversation, a medium where different processes can interact. Conversations are more expressive and more complex than SSCC sessions or streams, since they allow for direct multiparty interactions, while such a kind of interaction has to be programmed in some well-structured way in SSCC. As a consequence, static analysis in the Conversation Calculus is much more difficult, as can be seen by comparing its type system ensuring well-formed communications in [15] and our type system in Section 2.4, which has a similar aim.

Multiparty sessions, similar to conversations, have been studied also in μse [11]. There, two running multiparty sessions can dynamically merge. This makes static analysis even more difficult.

Two other calculi have been considered inside the Sensoria project: COWS [33] and SOCK [13]. Communication in these two calculi is not based on private sessions: they use instead the correlation set mechanism. Using this mechanism the recipient of a message is found based on (part of) the message content. For instance, a message including a specific user name is matched by the session created for managing all the communications for that specific user name. Correlation sets are used by the main technology for service-oriented orchestration: WS-BPEL [2]. However, since communication patterns depend on runtime values, they are more difficult to analyze statically. Only basic properties can be statically proved, as in [36]. In fact, private sessions can be seen as a constrained use of correlation sets ensuring more behavior predictability.

While both COWS and SOCK are based on correlation sets, they differ in many respects. In fact, COWS is more abstract, in the spirit of name passing calculi, while SOCK is essentially an imperative language extended with primitives for correlation-based communication. This makes SOCK more complex, but closer to real service-oriented languages. In fact, Jolie [27, 37], a full-fledged language for programming service-oriented applications, is based on the semantics of SOCK. We refer to [14] for a comparison of the calculi proposed inside the Sensoria project, and to [30] for a comparison of their behavioral theories.

Orc [35] is a language that focuses on orchestrating the concurrent invocation of services to achieve a given goal. The theory is built upon three composition operators—parallel composition, sequencing and selective pruning—while relying on a number of primitive services (sites, in the Orc's terminology) to perform basic computations. The symmetric parallel composition of f and g, written as f | g, permits independent computations and publications (i.e., outputs) from f and g. Sequencing of f and g, written as f >x >g, conveys *all* values x published by f to g. Selective pruning of f and g, written as g **where** x :∈ f conveys *some* value x published by f to g. We find that Orc does not provide enough support for orchestrating computations and does not support separation of concerns between orchestration and computation. In particular, and with respect to SSCC, Orc does not provide primitives for expressing conversations among services. This paper shows that flexible programming together with proofs of correctness for programming transformations can only be achieved in a language that takes conversation as a primitive concept.

Another thread of research [7, 16, 17, 26, 29] aims at capturing the principles behind Web service based business processes. A global description of communication behavior allows one to generate automatically an "endpoint-based" description of each participant to the protocol, a projection of the global scenario. We are at the same abstraction level of the endpoint calculus, but this one relies on multiparty session communication, and the considerations above for the Conversation Calculus apply.

1.5 The rest of this paper

We start by presenting the main ideas of the process language through two basic examples (Section 2.1). Then, Section 2.2 and Section 2.3 present, respectively, the syntax and the operational semantics (both a reduction-based system and a labeled transition system, shown to coincide on silent transitions) of the process description language.

Section 2.4 presents a first theoretical mechanism to study the (static) behavior of processes, a type system for SSCC. Equipped with this mechanism, we present in Section 3 more elaborate examples of systems specification in SSCC: Section 3.1 presents the encoding of common workflow patterns, and Section 3.2 presents a non-trivial programming exercise—the automotive case study.

Section 4 presents another theoretical mechanism to study the (dynamic) behavior of processes, giving notions of behavioral equivalences and proving some of their properties.

As an application of these theoretical results, Section 5 discusses how different programming styles applied to the same concrete problem may lead to seemingly different implementations, and shows, using the properties from the previous sections, that these implementations are indistinguishable if one abstracts from internal details. Furthermore, this section introduces a generic set of rules allowing to program SSCC processes according to different programming styles, and presents a set of (behavior-preserving) transformation rules allowing one to move from one style to another.

The last section summarizes the results obtained, and explores possible directions for future work.

2 SSCC: examples, syntax, and operational semantics

To motivate the basic constructs of the calculus, we present them one by one, incrementally building a simple example. A more elaborate example follows, to illustrate the expressive power of the calculus. Afterwards, we define rigorously the syntax, the operational semantics, and the type system of SSCC.

2.1 Modeling: basic examples

A first example. We start by defining a simple process to deliver the price of a room for a given date at a given hotel.

```
(date) ⟨query−the−hotel−db⟩ . price
```

Here, the parentheses in (date) indicate the reception of a value, and an identifier alone, as in price, means publishing a value. We use the notation ⟨ activity ⟩ to refer to code for some activity which is not (yet) specified. For instance, ⟨query−the−hotel−db⟩ stands for code which looks, in the hotel database, for the price of the stay; when running, this code uses an actual value for the parameter (date, in this example). Hotel bologna may turn the process above into a service definition, by writing:

```
bologna ⇒ (date) ⟨query−the−hotel−db⟩ . price
```

Here bologna is the name of a service that can be invoked to interact with the hotel, and whose behavior is as above. A client is supposed to meet the expectations of the service by providing a date and requesting a price:

```
bologna ⇐ 31Dec2012.(price) ⟨use−price⟩
```

Here we have an invocation (⇐) of service bologna, whose behavior first sends 31Dec2012 as date, then waits for an answer, which is stored in price. The code continues with some unspecified use of the received price.

When the service provider (⇒) and the service client (⇐) get together, by means, *e.g.*, of parallel composition, a *conversation* takes place, and values are exchanged in both directions.

Now suppose that a broker comes to the market trying to provide better deals for its clients. The behavior of the broker is as follows: it asks prices to three hotels that it knows of (bologna, lagoa, lisbon), waits for two results, and publishes the best offer of the two. Calling the three services for the same given date is as above:

```
bologna ⇐ date.(price1) ... |
lagoa ⇐ date.(price2) ... |
lisbon ⇐ date.(price3) ...
```

Note the use of parallel composition | to perform the three invocations concurrently.

In order to collect the prices for further processing, we introduce a *stream* constructor, playing the role of a *service orchestrator*. The various prices are fed into the stream; another process reads the stream. We write it as follows.

```
stream
  bologna ⇐ date.(price1).feed price1 |
  lagoa ⇐ date.(price2).feed price2 |
  lisbon ⇐ date.(price3).feed price3
as f in
  f(x).f(y).⟨publish−the−min−of−x−and−y⟩
```

To write price1 into a stream we use the syntax **feed** price1. To read a value from stream f we use f(x).⟨use−x⟩, where ⟨use−x⟩ is the code that uses the parameter x. Writing is an anonymous operation (feeds to the nearest enclosing stream), whereas reading is named. The above pattern is so common that we provide a special syntax for it. Inspired by Orc, **call** stands for an invocation of a (parametric) service, whereas $P >^{n} x_1 ... x_n > Q$ models the flow of n values from process P to process Q, via a stream. We refer to Figure 9 in page 15 for a formal definition of the derived syntactic constructs.

```
(call bologna(date) |
 call lagoa(date) |
 call lisbon(date)) >²
   x y > ⟨publish−the−min−of−x−and−y⟩
```

To complete the example we rely on a min service, chaining the first two answers, and publishing the result. We also turn the code into a service named broker.

```
broker ⇒ (date).(
  (call bologna(date) |
   call lagoa(date) |
   call lisbon(date)) >²
     x y > call min(x,y) >¹ m > m)
```

In detail, the first two values returned by the services, x and y, are used to invoke service min (note again the use of special syntax **call**). The value returned by service min is bound to m and returned to the client of service broker (the last occurrence of m is the return value).

Note that a client interacts with the broker using the same protocol that it would use to interact with a particular hotel named broker. The downside of this approach is that the client does not know which hotel offers the best price; it is not difficult to adapt the example to overcome this limitation.

Using **call** and $P >^{n} x_1 ... x_n > Q$ we have avoided explicitly mentioning streams altogether. Direct stream manipulation can however be quite handy. The following example shows a broker that logs all three answers by sending them to a specific service log, while publishing the best price of the first two (*cf.* the Discriminator Pattern [43] in Section 3.1).

```
stream ... as f in
  f(x).f(y).call min(x,y) >^1
    m > (m | f(z).log ⇐ x.y.z)
```

Our language is equipped with a notion of types,[1] allowing to statically filter programs that may incur in *conversation errors*, such as both the service provider and the client expecting a value at the same time (while if one is expecting a value the other one should be sending it), or the service expecting a value while the client has already terminated its execution. Returning to the hotel example, we can easily see that the conversation between the service provider (⇒) and the client (⇐) is, from the point of view of the provider, as follows: expect a date; send a price; terminate. The whole process of querying the hotel database to obtain the price is opaque to the client, and does not show up in the type. We write the type for an hotel as:

```
bologna :: [?Date.!Price.end]
```

The protocol with the broker is somewhat more complex, yet its interface with the client is exactly the same.

```
broker :: [?Date.!Price.end]
```

All values in a stream are required to be of the same type. The type of a process is a pair describing the conversation it engages into and the values it writes into its stream. Considering the part **stream** P **as** f **in** Q of the broker example, we have that P is of type (**end**, Price), meaning that P does not engage in any interaction with the client, and that it feeds Price values into the stream. On the other hand, Q is of type (!Price.**end**, T), since it communicates a price to the client (the type of the stream is arbitrary, given that Q does not feed into its stream).

A memory cell. Even if a memory cell is not a common scenario in SOC, stateful services are. Examples abound in the literature, from data-structures to weblog updates. Contrary to SCC [8], our language allows writing stateful services without exploiting service termination. Inspired by the encoding of objects in the π-calculus [39], we set up a simple, ephemeral, service to produce a value: buffer ⇒ v. The persistent service get below (persistent services are identified by the extra $*$ in the service definition) calls the buffer service to obtain its value (thus consuming the service provider), replies the value to the client, and replaces the buffer service.

```
get *⇒ call buffer >^1 v > (v | buffer ⇒ v)
```

The service set calls the buffer service (in order to consume the service provider), then gets the new value from the client and replaces the buffer with this value.

```
set *⇒ call buffer >^1 > (w)(buffer ⇒ w)
```

Note that above, in the operator $P >^1 > Q$, the name of the parameter is not written: we omit it since it is never used in the continuation.

Finally, the cell service sets up three services—get, set, and buffer—sends the first two to the client, and keeps buffer locally with initial value 0.

```
cell *⇒ (νbuffer,get,set).get.set.
 (buffer ⇒ 0 |
  get *⇒ call buffer >^1 v >
      (v | buffer ⇒ v) |
  set *⇒ call buffer >^1 > (w)(buffer ⇒ w))
```

[1]Types and the type system are presented in detail in Section 2.4.

$P, Q \quad ::=$		*Processes*		
	$\mid$ $\mathbf{0}$	Terminated process		
	$\mid$ $P \mid Q$	Parallel composition	$\mid$ $(\nu\, a)P$	Name restriction
	$\mid$ X	Process variable	$\mid$ $\textbf{rec}\, X.P$	Recursive process definition
	$\mid$ $a \Rightarrow P$	Service definition	$\mid$ $a \Leftarrow P$	Service invocation
	$\mid$ $v.P$	Value sending	$\mid$ $(x)P$	Value reception
	$\mid$ $\textbf{stream}\, P\, \textbf{as}\, f\, \textbf{in}\, Q$	Stream	$\mid$ $\textbf{feed}\, v.P$	Feed the process' stream
	$\mid$ $f(x).P$	Read from a stream		
$v \quad ::=$		*Values*		
	a	Service name	$\mid$ $\textbf{unit}$	Unit value

Figure 1: The syntax of SSCC

2.2 Syntax

The syntax for SSCC processes is inspired by that of the π-calculus, but it includes several additional features. A process may post/read a message to/from its environment; more complex processes may be built by joining two processes in parallel, restricting a name within a process or using recursion (guarded by prefixes).

Furthermore, a process may be a service definition $a \Rightarrow P$ or service invocation $a \Leftarrow P$, containing the name a of the service and a protocol P. When a service invocation meets its corresponding definition, the corresponding protocols execute within a session whose name is freshly generated, and messages posted by one endpoint are read by the other endpoint of that session.

Service orchestration is achieved by means of stream composition. A stream is a one-way static channel whose name cannot be communicated; the process "inside" a stream can only post values to that stream, whereas the "outer" process can only read from that stream. It is important to point out that any process may (directly) write to exactly one stream, but might be able to read from several; hence the need to name the stream in the reading, but not in the writing, context. This choice simplifies the type system and the behavioral analysis, not compromising expressiveness, as far as we noticed with the (large amount of) examples developed. Intuitively, communication via streams corresponds to local interaction, whereas service interaction is meant for remote communication.

Notation. Processes are built using three kinds of identifiers: *service names*, *stream names*, and *process variables*. Service names are ranged over by a and b; values, ranged over by u and v, can be either service names or the **unit** value. Basic values such as integers and strings can be easily added, and will be used in examples. Names for values can also be used as variables (bound by value reception or read from stream), and we use x and y in this case. Stream names are ranged over by f and g. Process variables are ranged over by X and Y, and are used to define recursive processes. All process variables are required to be bound, *i.e.*, they must appear only within a recursive process definition of which they are the recursion variable. Thus, we consider a process well-formed only if it contains no free process variables.

Definition 1 (Syntax)**.** *The grammar in Figure 1 defines the* syntax of SSCC processes.

The first five cases of the grammar introduce standard process calculi operators: the terminated process **0**, parallel composition P|Q, name restriction (notice that only service names can be restricted), and recursion (to define recursion we need both process variables X and recursive process definitions **rec** X.P).

We then have two constructs to *build services*: definition (or provider) and invocation (or client). They are both defined by their name a and protocol P. Service definition and service invocation are symmetric, differently from SCC [8]. *Service protocols* are built using value sending and receiving, allowing bidirectional communication between clients and servers.

Finally, there are the three constructs for *service orchestration*, which constitute the main novelty of our calculus. The stream construct declares a stream f for communication from P to Q. P can insert a value v

$P, Q \ ::=$	*Runtime processes*		
$\dots$	as in Figure 1		
$\mid\ r \rhd P$	Server session	$\mid\ r \lhd P$	Client session
$\mid\ (\nu\, r)P$	Session restriction	$\mid\ \textbf{stream}\,P\,\textbf{as}\,f = \vec{v}\,\textbf{in}\,Q$	Stream with values

Figure 2: The runtime syntax of SSCC

into the stream f using **feed** v.P', and Q can read from there using f(x).Q'. Streams can be considered either ordered or unordered. An unordered stream is a multiset, while an ordered one is a queue. In most cases the difference is not important. Herein, streams are considered ordered, acting as queues. Note that stream communication is inherently asynchronous. We write w :: $\vec{v}$ for the stream obtained by adding w to stream $\vec{v}$, and $\vec{v}$ :: w for a stream from which w can be removed. In the latter case $\vec{v}$ is what we get after removing w. We denote the empty stream by $\langle\rangle$.

Observe that these processes do not yet contain sessions. Indeed, sessions can only arise as a result of the interaction of a service definition and a service invocation, which produces an active session. Therefore, an extended run-time syntax is needed. This syntax distinguishes a fourth set of identifiers: *session names*, ranged over by r and s. The letters n and m are used to range over both session names and service names. Moreover, at runtime, values in the stream are stored together with the stream definition: the static construct **stream** P **as** f **in** Q can be seen as an abbreviation of **stream** P **as** f=$\langle\rangle$ **in** Q.

Definition 2 (Runtime syntax)**.** *The grammar in Figure 2 defines the* syntax of runtime SSCC processes.

Henceforth, we only consider processes derived from the ones in the static syntax.

2.3 Operational semantics

Since the aim of SSCC is to model systems that change over time, it is essential to describe how they evolve. This is done by giving an operational semantics, which is a reduction relation on runtime processes.

Notation Some constructs act as binders. Name x is bound in $(x)P$ and in $f(x).P$; name n is bound in $(\nu\, n)P$; stream f is bound in $\textbf{stream}\,P\,\textbf{as}\,f = \vec{v}\,\textbf{in}\,Q$ with scope Q; and process variable X is bound in $\textbf{rec}\,X.P$. The sets of free and bound names in P are denoted, respectively, by $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$. The set of names in P is $\mathrm{n}(P) = \mathrm{fn}(P) \cup \mathrm{bn}(P)$. If $\vec{w} = w_1 \cdots w_n$ with $n \geq 0$, then $\mathrm{Set}(\vec{w}) = \{w_1, \dots, w_n\}$. We work up to α-conversion and follow Barendregt's variable convention, whereby all variables in binding occurrences in any mathematical context are pairwise distinct and distinct from the free variables [5]. We use capture avoiding substitutions (thanks to Barendregt convention) to replace names for names, as in $[{}^{v}\!/\!{}_{x}]$, and processes for process variables, as in $[{}^{\textbf{rec}\,X.P}\!/\!{}_{X}]$.

Structural congruence. Before proceeding further, it is appropriate to introduce a structural congruence relation for SSCC processes. In the rules in Figure 3 and in the following, in processes containing sessions, $r \bowtie P$ stands for either $r \lhd P$ or $r \rhd P$; multiple occurrences of $r \bowtie P$ within the same rule are instantiated in the same way (*i.e.*, all $r \lhd P$ or all $r \rhd P$), while occurrences of $r \,\overline{\bowtie}\, P$ stand for the opposite instantiation (*i.e.*, where $r \bowtie P$ stands for $r \lhd P$, $r \,\overline{\bowtie}\, P$ stands for $r \rhd P$, and vice-versa).

Definition 3 (Structural congruence relation)**.** *The* structural congruence relation $\equiv$ on runtime SSCC processes *is the smallest congruence closed under the rules in Figure 3.*

Reduction relation. Interactions can happen in different active contexts. Since our interactions are binary, we introduce also two-holed contexts, which we call double contexts.

Definition 4 (Active contexts)**.** *The grammar in Figure 4 generates* active contexts $\mathcal{C}[\![]\!]$ *and* double contexts $\mathcal{D}[\![,]\!]$.

Applying a double context to two processes P_1 and P_2 produces the process obtained by replacing the first (in the prefix visit of the syntax tree) hole $\bullet$ with P_1 and the second hole $\bullet$ with P_2.

$$
\begin{array}{ll}
P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R) & (\text{S-NIL, S-COMM, S-ASSOC}) \\
(\nu\, n)P|Q \equiv (\nu\, n)(P|Q) \quad \text{if } n \notin \text{fn}(Q) \qquad r \bowtie (\nu\, a)P \equiv (\nu\, a)(r \bowtie P) & (\text{S-EXTR-PAR, S-EXTR-SESS}) \\
\textbf{stream}\,(\nu\, a)P\,\textbf{as}\, f = \vec{v}\,\textbf{in}\, Q \equiv (\nu\, a)(\textbf{stream}\, P\,\textbf{as}\, f = \vec{v}\,\textbf{in}\, Q) \quad \text{if } a \notin \text{fn}(Q) \cup \text{Set}(\vec{v}) & (\text{S-EXTR-STREAML}) \\
\textbf{stream}\, P\,\textbf{as}\, f = \vec{v}\,\textbf{in}\,(\nu\, a)Q \equiv (\nu\, a)(\textbf{stream}\, P\,\textbf{as}\, f = \vec{v}\,\textbf{in}\, Q) \quad \text{if } a \notin \text{fn}(P) \cup \text{Set}(\vec{v}) & (\text{S-EXTR-STREAMR}) \\
\textbf{stream}\, P\,\textbf{as}\, f\,\textbf{in}\, Q \equiv \textbf{stream}\, P\,\textbf{as}\, f = \langle\rangle\,\textbf{in}\, Q & (\text{S-RUN-TIME-STREAM}) \\
(\nu\, n)(\nu\, m)P \equiv (\nu\, m)(\nu\, n)P \qquad (\nu\, a)\mathbf{0} \equiv \mathbf{0} \qquad \textbf{rec}\, X.P \equiv P[\textbf{rec}\, X.P/X] & (\text{S-SWAP, S-COLLECT, S-REC})
\end{array}
$$

Figure 3: The structural congruence relation

$$
\begin{array}{rcl}
\mathcal{C}⟦⟧ & ::= & \bullet \mid \mathcal{C}⟦⟧|Q \mid P|\mathcal{C}⟦⟧ \mid (\nu\, n)\mathcal{C}⟦⟧ \\
& & \mid \textbf{stream}\,\mathcal{C}⟦⟧\,\textbf{as}\, f = \vec{v}\,\textbf{in}\, Q \mid \textbf{stream}\, P\,\textbf{as}\, f = \vec{v}\,\textbf{in}\,\mathcal{C}⟦⟧ \mid r \bowtie \mathcal{C}⟦⟧ \\
\mathcal{D}⟦,⟧ & ::= & \mathcal{C}'⟦⟧|\mathcal{C}''⟦⟧ \mid \textbf{stream}\,\mathcal{C}'⟦⟧\,\textbf{as}\, f = \vec{v}\,\textbf{in}\,\mathcal{C}''⟦⟧
\end{array}
$$

Figure 4: Active and double contexts

Definition 5 (Reduction semantics)**.** *The rules in Figure 5, together with symmetric rules of* R-COMM *and of* R-SYNC *(swapping the processes in the two holes of $\mathcal{D}⟦,⟧$), inductively define the* reduction relation *on* SSCC *processes.*

Rule R-SYNC allows a service invocation and a service definition to interact. This interaction produces a pair of complementary sessions, distinguished by a fresh restricted name r. Notice that both the service invocation and the service definition disappear. Rule R-COMM allows communication between corresponding sessions. Since value sending and receiving refer to the innermost session, we have to ensure that r is such a session. This is done by requiring that $\mathcal{C}⟦⟧$ and $\mathcal{C}'⟦⟧$ do not contain sessions around the hole. Then there are the two rules dealing with streams: rule R-FEED puts a value in the stream while rule R-READ takes a value from the stream. In R-FEED, similarly to what happens in R-COMM, the right premise ensures that the value sent by the feed operation is not captured by an inner stream construct in context $\mathcal{C}⟦⟧$. Finally rule R-CONG allows reductions to happen inside arbitrary active contexts, and rule R-STR exploits structural congruence.

2.4 Type system

This section deals with the static behavior of SSCC processes. Type systems provide protocol information, abstracting from the actual data being sent, and are powerful enough to prove a number of interesting properties about termination.

Typing for SSCC processes is a way of providing information about their protocol, telling the world about the data they are expecting and how they communicate with the outside. This simple type system, built along the lines of session types [22, 25, 42, 46], is strong enough to ensure protocol compatibility among clients and servers. It is also able to deal with many different interacting services simultaneously.

Definition 6 (Types)**.** *The grammar in Figure 6 defines the* syntax of types.

Types are divided into three classes. Types for *values T* are either **Unit**, which denotes the only basic type,[2] or $[U]$, which is the type of a service or session with protocol U. This protocol is always seen from

[2]To be possibly extended with, say, integers and strings.

$$\frac{\mathcal{D}[\![\,,\,]\!] \text{ does not bind } r \text{ or } a \qquad r \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q) \cup \mathrm{fn}(\mathcal{D}[\![\,,\,]\!])}{\mathcal{D}[\![a \Rightarrow P, a \Leftarrow Q]\!] \to (\nu\, r)\mathcal{D}[\![r \rhd P, r \lhd Q]\!]} \qquad (\text{R-SYNC})$$

$$\frac{\begin{array}{c}\mathcal{D}[\![\,,\,]\!], \mathcal{C}[\![\,]\!], \text{ and } \mathcal{C}'[\![\,]\!] \text{ do not bind } r \text{ or } v \\ \mathcal{C}[\![\,]\!] \text{ and } \mathcal{C}'[\![\,]\!] \text{ do not contain sessions around the hole}\end{array}}{(\nu\, r)\mathcal{D}[\![r \bowtie \mathcal{C}[\![v.P]\!], r \,\overline{\bowtie}\, \mathcal{C}'[\![(x)Q]\!]]\!] \to (\nu\, r)\mathcal{D}[\![r \bowtie \mathcal{C}[\![P]\!], r \,\overline{\bowtie}\, \mathcal{C}'[\![Q[{}^{v}\!/\!{}_{x}]]\!]]\!]} \qquad (\text{R-COMM})$$

$$\frac{\mathcal{C}[\![\,]\!] \text{ does not bind } w \qquad \text{the hole does not occur in } \mathcal{C}[\![\,]\!] \text{ in the left part of a stream context}}{\textbf{stream}\, \mathcal{C}[\![\textbf{feed}\, w.P]\!] \,\textbf{as}\, f = \vec{v} \,\textbf{in}\, Q \to \textbf{stream}\, \mathcal{C}[\![P]\!] \,\textbf{as}\, f = w :: \vec{v} \,\textbf{in}\, Q} \qquad (\text{R-FEED})$$

$$\frac{\mathcal{C}[\![\,]\!] \text{ does not bind } w \text{ or } f}{\textbf{stream}\, P \,\textbf{as}\, f = \vec{v} :: w \,\textbf{in}\, \mathcal{C}[\![f(x).Q]\!] \to \textbf{stream}\, P \,\textbf{as}\, f = \vec{v} \,\textbf{in}\, \mathcal{C}[\![Q[{}^{w}\!/\!{}_{x}]]\!]} \qquad (\text{R-READ})$$

$$\frac{P \to P'}{\mathcal{C}[\![P]\!] \to \mathcal{C}[\![P']\!]} \qquad \frac{Q \equiv P \to P' \equiv Q'}{Q \to Q'} \qquad (\text{R-CONG, R-STR})$$

Figure 5: The reduction relation

T	$::=$	*Types*
	Unit	unit type
	$\mid\ [U]$	service type
U	$::=$	*Conversation types*
	$?T.U$	input
	$\mid\ !T.U$	output
	$\mid$ **end**	end of conversation
	$\mid\ \alpha$	type variable
	$\mid$ **rec** $\alpha.U$	recursive type

Figure 6: The syntax of value types

the server point of view, regardless of the role of the process being typed. Types for *streams* are of the form $\langle T \rangle$ where T is the type of the values the stream carries. Finally, types for *processes* are of the form (U,T) where U is the protocol that the process follows, and T is the type of the values the process feeds into its stream.

The **rec** operator for types is a binder, giving rise, in the standard way, to notions of bound and free variables and α-equivalence. As was the case for processes, α-convertible types are considered the same. Furthermore, this paper takes an *equi-recursive* view of types, not distinguishing between a type **rec** $\alpha.U$ and its unfolding $T[{}^{\textbf{rec}\ \alpha.U}\!/\!{}_{\alpha}]$. We are interested on *contractive* (not including subterms of the form **rec** α.**rec** $\alpha_1 \ldots$ **rec** $\alpha_n.\alpha$) types only; see [38].

In order to establish communication, two processes must have complementary protocols, in the sense that one is ready to send values of the type that the other is expecting, and vice-versa. This is captured via the operation of complementation on conversation types, defined in Figure 7. Intuitively, if a client executes protocol U and a server implements protocol $\overline{U}$, then the conversation between them can proceed without errors.

Typing judgments are as follows:

$$\Gamma \vdash P \colon (U,T) \qquad \textit{Processes}$$
$$\Gamma \vdash v \colon T \qquad \textit{Values}$$

where Γ is a map with entries $a\colon T$, $r\colon T$, $f\colon \langle T \rangle$, and $X\colon (U,T)$.

$$\overline{?T.U} \triangleq\, !T.\overline{U} \qquad \overline{!T.U} \triangleq ?T.\overline{U} \qquad \overline{\textsf{end}} \triangleq \textsf{end} \qquad \overline{\alpha} \triangleq \alpha \qquad \overline{\textsf{rec}\ \alpha.U} \triangleq \textsf{rec}\ \alpha.\overline{U}$$

Figure 7: The complement of a conversation type

$$\Gamma, n\colon T \vdash n\colon T \qquad \Gamma, f\colon \langle T\rangle \vdash f\colon \langle T\rangle \qquad \Gamma \vdash \textsf{unit}\colon \textsf{Unit} \qquad \text{(T-NAME, T-SNAME, T-UNIT)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash v\colon T'}{\Gamma \vdash v.P\colon (!T'.U,T)} \qquad \frac{\Gamma, x\colon T' \vdash P\colon (U,T)}{\Gamma \vdash (x)P\colon (?T'.U,T)} \qquad \text{(T-SEND, T-RECEIVE)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [U]}{\Gamma \vdash a \Rightarrow P\colon (\textsf{end},T)} \qquad \frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash a\colon [\overline{U}]}{\Gamma \vdash a \Leftarrow P\colon (\textsf{end},T)} \qquad \text{(T-DEF, T-CALL)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash r\colon [U]}{\Gamma \vdash r \rhd P\colon (\textsf{end},T)} \qquad \frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash r\colon [\overline{U}]}{\Gamma \vdash r \lhd P\colon (\textsf{end},T)} \qquad \text{(T-SESS-S, T-SESS-C)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash v\colon T}{\Gamma \vdash \textsf{feed}\, v.P\colon (U,T)} \qquad \frac{\Gamma, x\colon T \vdash P\colon (U,T') \quad \Gamma \vdash f\colon \langle T\rangle}{\Gamma \vdash f(x).P\colon (U,T')} \qquad \text{(T-FEED, T-READ)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma \vdash Q\colon (\textsf{end},T)}{\Gamma \vdash P|Q\colon (U,T)} \qquad \frac{\Gamma \vdash P\colon (\textsf{end},T) \quad \Gamma \vdash Q\colon (U,T)}{\Gamma \vdash P|Q\colon (U,T)} \qquad \text{(T-PAR-L, T-PAR-R)}$$

$$\frac{\Gamma \vdash P\colon (U,T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (\textsf{end},T') \quad w \in \mathrm{Set}(\vec{v}) \Rightarrow \Gamma \vdash w\colon T}{\Gamma \vdash \textsf{stream}\, P\, \textsf{as}\, f = \vec{v}\, \textsf{in}\, Q\colon (U,T')} \qquad \text{(T-STREAM-L)}$$

$$\frac{\Gamma \vdash P\colon (\textsf{end},T) \quad \Gamma, f\colon \langle T\rangle \vdash Q\colon (U,T') \quad w \in \mathrm{Set}(\vec{v}) \Rightarrow \Gamma \vdash w\colon T}{\Gamma \vdash \textsf{stream}\, P\, \textsf{as}\, f = \vec{v}\, \textsf{in}\, Q\colon (U,T')} \qquad \text{(T-STREAM-R)}$$

$$\frac{\Gamma, X\colon (U,T) \vdash P\colon (U,T)}{\Gamma \vdash \textsf{rec}\, X.P\colon (U,T)} \qquad \frac{\Gamma, n\colon T_1 \vdash P\colon (U,T)}{\Gamma \vdash (\nu n)P\colon (U,T)} \qquad \text{(T-REC, T-RES)}$$

$$\Gamma, X\colon (U,T) \vdash X\colon (U,T) \qquad \Gamma \vdash \mathbf{0}\colon (\textsf{end},T) \qquad \text{(T-VAR, T-NIL)}$$

Figure 8: The type system

Definition 7 (Type system)**.** *The rules in Figure 8 inductively define the type system for* SSCC *processes.*

The type of a process is an abstraction of its behavior: its first component shows the protocol of the process, while the second component traces the type of the values fed to its stream. Notice that the properties of internal sessions and streams are guaranteed by the typing derivation and the typing assumptions in Γ, and they do not influence the type of the process. For instance, if the process is a session r $\rhd$ P, then its protocol is **end**, but the protocol followed by P is traced by an assumption $r\colon [U]$ in Γ. When the complementary session is found, the compatibility check is performed.

These types force protocols to be sequential, which the authors consider to be a good programming style. Suppose for instance that the protocol contained two parallel outputs: then there should be two inputs in the complementary protocol, and we cannot know which output is matched with each input. Either this is irrelevant for the process, and the outputs can be sorted out in an arbitrary way, or it is relevant, and it should generate an error.

Another aspect is that parallel protocols are more complex to check for compatibility. Notice that this does not forbid having, *e.g.*, two concurrent service invocations, since sequentiality is only enforced inside protocols.

SSCC equipped with this type system is type safe: typable processes are not errors, nor can they generate errors. The proof of this result requires, as usual, a type preservation property—subject reduction—and a definition of erroneous processes. The proofs of the theorems below can be found in Appendix A.

Theorem 1 (Subject reduction)**.** *Let* $\Gamma \vdash P\colon (U,T)$ *and* $P \to P'$. *Then* $\Gamma \vdash P'\colon (U,T)$.

Theorem 2 (Type Safety)**.** *Let* P *be a typable process. Then* P *has no subterm of the following forms.*

Protocol:

$\mathcal{D}[\![r \bowtie \mathcal{C}[\![v.P]\!], r \mathrel{\overline{\bowtie}} \mathcal{C}'[\![u.Q]\!]]\!]$	*Two outputs*
$\mathcal{D}[\![r \bowtie \mathcal{C}[\![v.P]\!], r \mathrel{\overline{\bowtie}} \mathbf{0}]\!]$	*Output and* **0**
$\mathcal{D}[\![r \bowtie \mathcal{C}[\![(x)P]\!], r \mathrel{\overline{\bowtie}} \mathcal{C}'[\![(y)Q]\!]]\!]$	*Two inputs*
$\mathcal{D}[\![r \bowtie \mathcal{C}[\![(x)P]\!], r \mathrel{\overline{\bowtie}} \mathbf{0}]\!]$	*Input and* **0**

where in all the cases $\mathcal{D}[\![,]\!]$ *does not bind* r*, and* $\mathcal{C}[\![]\!]$ *and* $\mathcal{C}'[\![]\!]$ *do not contain sessions around the* $\bullet$*.*

Sequentiality:

$\mathcal{D}[\![v.P, u.Q]\!]$	*Parallel outputs*
$\mathcal{D}[\![(x)P, u.Q]\!]$	*Parallel input and output*
$\mathcal{D}[\![v.P, (y)Q]\!]$	*Parallel output and input*
$\mathcal{D}[\![(x)P, (y)Q]\!]$	*Parallel inputs*

where in all cases $\mathcal{D}[\![,]\!]$ *does not contain sessions around the* $\bullet$*.*

An example of *protocol failure* is illustrated by process r ▷ v.P | r ◁ nil. This process cannot be typed, since the two parallel components require different assumptions for r, namely r: [!T.U'] where T is the type of v, and r: **end**, respectively. Similarly, a *non-sequential conversation* is r ▷ (v.P | u.Q), which also cannot be typed, since both v.P and u.Q have non-**end** protocols, thus forbidding the application of rules for parallel composition. Techniques used for session types can be adapted to *type check* SSCC processes [44].

As an example we show the typing judgment for the protocol of the memory cell from Section 2.1. Services in SSCC are ephemeral: they do not survive invocation. Recursion can be used to provide persistent services: a service a ⇒ P can be made persistent by writing instead **rec** X.a ⇒ (P | X), which we abbreviate as a ∗⇒ P (see Figure 9).

```
get :: [!Int.end]
get *⇒ call buffer >1 v > (v | buffer ⇒ v)
```

The type captures the fact that the get service publishes the integer stored in the memory cell.

```
set :: [?Int.end]
set *⇒ call buffer >1 > (w)(buffer ⇒ w)
```

The type indicates that a value is sent to the service (it will be used for updating the memory cell).

```
cell :: [![!Int.end].![?Int.end].end]
cell *⇒ (ν buffer, get, set).get.set.(buffer ⇒ 0 |
  get *⇒ call buffer >1 v > (v | buffer ⇒ v) |
  set *⇒ call buffer >1 > (w)(buffer ⇒ w))
```

The type for the Cell service is more interesting. It makes apparent the fact that, upon instantiation, the Cell service will publish two services, the first able to send a value (for getting the value of the memory cell) and the second able to receive a value (for setting the memory cell).

3 Specifying in SSCC

This section explores examples that highlight the versatility of SSCC, while comparing to solutions written in SCC and Orc. These two languages are briefly reviewed in Section 1.4. For modeling we use a few useful abbreviations, which are gathered in Figure 9. We will continue to use them in the rest of the paper.

The first example shows that naming streams can be handy. *Fork-join* is a pattern that spawns two threads, and resumes computation after receiving a value from each thread. In the example below, services a and b are run in parallel; **call** a feeds the first result produced by the service into stream f, and similarly for **call** b and stream g.

```
fork-and-join :: [?[!T1.end].?[!T2.end].!T1.!T2.end]
fork-and-join *⇒ (a)(b)(
  stream call a as f in
    stream call b as g in
      f(x).g(y).x.y)
```

$$\begin{array}{rcl}
\textbf{call}\ \mathsf{a}(\mathsf{x}_1, \ldots, \mathsf{x}_n) & \triangleq & \mathsf{a} \Leftarrow \mathsf{x}_1 \ldots \mathsf{x}_n.(\mathsf{y})\ \textbf{feed}\ \mathsf{y} \\
\mathsf{P} >^{\mathbf{n}} \mathsf{x}_1 \ldots\ \mathsf{x}_n > \mathsf{Q} & \triangleq & \textbf{stream}\ \mathsf{P}\ \textbf{as}\ \mathsf{f}\ \textbf{in}\ \mathsf{f}(\mathsf{x}_1).\ \ldots\ \mathsf{f}(\mathsf{x}_n).\mathsf{Q} \\
\mathsf{P} > \mathsf{x} > \mathsf{Q} & \triangleq & \textbf{stream}\ \mathsf{P}\ \textbf{as}\ \mathsf{f}\ \textbf{in}\ \textbf{rec}\ \mathsf{X}.\mathsf{f}(\mathsf{x}).(\mathsf{Q} \mid \mathsf{X}) \\
\mathsf{a} *\!\!\Rightarrow \mathsf{P} & \triangleq & \textbf{rec}\ \mathsf{X}.\ \mathsf{a} \Rightarrow (\mathsf{P} \mid \mathsf{X}) \\
\textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P} & \triangleq & \mathsf{b} \Leftarrow (\mathsf{x})(\mathsf{y})\ \mathsf{x} \Leftarrow \textbf{feed unit} >^{\mathbf{1}} > \mathsf{P} \\
\textbf{if}\ \neg\mathsf{b}\ \textbf{then}\ \mathsf{P} & \triangleq & \mathsf{b} \Leftarrow (\mathsf{x})(\mathsf{y})\ \mathsf{y} \Leftarrow \textbf{feed unit} >^{\mathbf{1}} > \mathsf{P} \\
\textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P}\ \textbf{else}\ \mathsf{Q} & \triangleq & \textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{P} \mid \textbf{if}\ \neg\mathsf{b}\ \textbf{then}\ \mathsf{Q} \\
\mathsf{T}_1 \to \ldots \to \mathsf{T}_n \to \mathsf{T} & \triangleq & [?\mathsf{T}_1 \ldots ?\mathsf{T}_n.!\mathsf{T}.\textbf{end}] \\
\varepsilon \to \mathsf{T} & \triangleq & [!\mathsf{T}.\textbf{end}] \\
\mathsf{Bool} & \triangleq & [![\textbf{end}].![\textbf{end}].\textbf{end}]
\end{array}$$

Assume omnipresent the following services,

```
tt *⇒ 0
true *⇒ tt.ff.0
false *⇒ ff.tt.0
```

and omniabsent the service for ff.

Figure 9: Derived constructs

The example is inspired in Orc [28,35], but Orc, when one of the service invocations at a or b completes, kills the other. Instead, we let them run to completion. Orc is not able to match our semantics: reading a single value from an expression can only be performed via the **where** construct, and that necessarily means terminating the evaluation of the expression. We feel that termination should be distinct from normal orchestration; we leave for further work termination (and the corresponding compensation). Notice, however, that the declared type makes sure that services a and b produce each a single value.

The second example describes an idiom where for each value x produced by a process P, a second process Q is started. If process P produces its values by feeding into its stream, then, in the process below, a new copy of process Q is spawned for each value read from the stream. Process

```
stream P as f in rec X.f(x).(Q | X)
```

can be abbreviated to P >x >Q (x can be dropped if it does not occur in Q), so that a service that reads news from sites CNN and BBC and e-mails each to a given address can be written as:

```
email−news :: [?Address.end]
email−news *⇒ (a)((call CNN | call BBC) > x > email ⇐ a.x)
```

The example and the short syntax is again from Orc. In this case we are faithful to the Orc semantics.

3.1 Workflow patterns

In this section we illustrate the expressiveness of SSCC by implementing the Workflow Patterns (WP) from van der Aalst et al. [43]. This permits contrasting again our approach with that of SCC and of Orc [19], which have similar aims. While WPs are an interesting benchmark, they are aimed at workflow description languages, not at calculi for SOC. For this reason some of the patterns are not meaningful (*e.g.*, WP11: Implicit Termination) in our context, while others are redundant (*e.g.*, WP12: Multiple Instances without Synchronization is similar to WP2: Parallel Split, since process calculi can obviously handle multiple instances). Also, some patterns require the ability to kill processes, which has not yet been introduced in SSCC, and thus are out of our possibilities. On the contrary WPs consider only "activities", *i.e.*, services that receive one value and give back one result, while our calculus can also model more complex protocols.

We describe all patterns (in reference [43]) as services; we also present their types. Those that have multiple entry points (the various merges, for example) are modeled with a vector of boolean values, describing which services should be invoked.

For us, an *activity* is a service that writes at most one value on the client side (replies at most one value). The simplest activity is the null service.

```
nullService :: ε → Unit
nullService ⤃ unit
```

Most of the patterns below allow definitions in SSCC that do not directly use either the stream operations (**stream**, **feed**, and f(x).P) or recursion.

In what follows we give a brief description of each workflow pattern and present an illustrative example, both taken from [43]. To allow a comparison we also show how the patterns can be implemented in SCC. Services in SCC have always one parameter: we exploit it as first input for the server if the server protocol should start with an input, otherwise we assume it is unused and use **unit** as invocation value. We concentrate on WP1, WP3, WP4, WP9, WP10, and WP17, the patterns we find more significant and which better highlight the expressive power of SSCC. The reader should refer to Appendix B for the modeling of all the workflow patterns from [43].

WP1: Sequence

"An activity in a workflow process is enabled after the completion of another activity in the same process. Example: an insurance claim is evaluated after the client's file is retrieved."

```
seq :: (ε → T₁) → (ε → T₂) → T₂
seq ⤃ (a₁)(a₂)call a₁ >¹> call a₂ >¹ x > x
```

In Orc the implementation is similar. In SCC the most direct implementation is:

```
seq ⇒ (a₁)(a₂) a₂ ⇐ a₁ ⇐ unit
```

This implementation is fine for activities (actually here a_2 is invoked with the value from a_1 rather than with **unit**: this problem is solved in SSCC), but if a_1 is not an activity then a_2 is called for each value returned by a_1, and this is not the expected semantics. In SSCC this cannot happen, since the remaining values returned by a_1 stay forever in the stream. To enforce correct behavior in SCC one should write:[3]

```
seq ⇒ (a₁)(a₂)(νr)(r ▷ (a₁ ⇐ unit | (res) return res) |
                    r̄ ▷ (v) a₂ ⇐ unit)
```

Notice that the SCC encoding above uses a conversation (a process of the form $r \rhd P \mid \bar{r} \lhd Q$), which we view as runtime syntax in SSCC. However, sessions can be also avoided in SCC using "fake" service invocations and definitions (nevertheless, service definitions stay there afterward, since they are persistent).

WP3: Synchronization

"A point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once. Example: insurance claims are evaluated after the policy has been checked and the actual damage has been assessed."

```
sync :: (ε → T) → ... → (ε → T) → Unit
sync ⤃ (a₁)...(aₙ)(call a₁ | ... | call aₙ) >ⁿ> unit
```

Orc uses the **where** operator, and SCC uses sessions (or "fake services"):

```
sync ⇒ (a₁)...(aₙ)(νr)(r ▷ (a₁ ⇐ unit | ... | aₙ ⇐ unit) |
                           r̄ ▷ (x₁)...(xₙ) return unit)
```

[3]This can be done also by type checking the protocol for a_1: this feature is not yet available in SCC, but it can be easily transferred there.

WP4: Exclusive Choice

“A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen. Example: based on the workload, a processed tax declaration is either checked using a simple administrative procedure or is thoroughly evaluated by a senior employee.”

```
xor :: Bool → (ε → T) → (ε → T) → T
xor *⇒ (b)($a_1$)($a_2$) if b then call $a_1$ $>^1$ x > x else call $a_2$ $>^1$ x > x
```

Notice that **if** – **then** – **else** cannot be typed with the current type system unless both branches have the empty protocol (since they occur in parallel). This problem can be solved by adding a dedicated typing rule to exploit the knowledge that only one of the branches is actually executed.

In SCC one can implement true and false in a similar way. Then we have:

```
if b then P = (νs) s ▷ b {(−) (x)(y) return x} ⇐ unit
                | $\bar{s}$ ▷ ($x_1$) (νr) r ▷ $x_1$ {(−)return unit} ⇐ unit
                                  | $\bar{r}$ ▷ (z) P
if ¬b then P = (νs) s ▷ b {(−) (x)(y) return y} ⇐ unit
                | $\bar{s}$ ▷ ($x_1$) (νr) r ▷ $x_1$ {(−)return unit} ⇐ unit
                                  | $\bar{r}$ ▷ (z) P
```

Notice that while in SSCC feeds from P are not intercepted by the if context, in SCC the returns are lost, since P is executed inside a subsession. To forward the results to the caller, extra programming effort is required. Actually, since P cannot be executed at top level (since in order to start it when a trigger coming from a subsession is received, the trigger should be transmitted using a return, which either goes to the other side, or executes P inside a session) a forward of values is needed, but in SCC one is able to specify only a finite amount of forwarding. Thus, if P can give back an unbounded number of replies, this cannot be programmed. Nevertheless, in the following example we always suppose to have the **if** with forwarding of the results. Since workflow patterns deal only with activities (one result), then the forwarding can be implemented.

The **if** – **then** – **else** is as in SSCC.

```
xor ⇒ (b)($a_1$)($a_2$) if b then $a_1$ ⇐ unit else $a_2$ ⇐ unit
```

Because of the above observation the xor in SCC (with the above implementation of **if**) gives back no value.

WP9: Discriminator

“The discriminator is a point in a workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and “ignores” them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again (which is important, otherwise it could not really be used in the context of a loop). Example: to improve query response time, a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.”

```
discriminator :: (ε → T) →...→ (ε → T) → Unit
rec X.discriminator ⇒ ($a_1$)...($a_n$)
                       stream call $a_1$ | ... | call $a_n$ as f in
                       f($x_1$).unit.f($x_2$)...f($x_n$).X
```

In SCC, we cannot control the point where the service discriminator becomes available again.

```
discriminator ⇒ ($a_1$)...($a_n$)
                   (νr) r ▷ $a_1$ ⇐ unit | ... |$a_n$ ⇐ unit
                        $\bar{r}$ ▷ ($x_1$).return unit.($x_2$)...($x_n$)
```

Here, the Orc implementation supposes the existence of a basic site S, with methods put and get, acting as a buffer. This site cannot be described in Orc (Orc does not deal with site programming). We think that sites should deal only with computation, while all the coordination should be done at the coordination language level. This implementation fails to satisfy this separation of concerns principle. We are not aware of better implementations in Orc.

WP10: Arbitrary Cycles

"A point in a workflow process where one or more activities can be done repeatedly." Arbitrary cycles can be obtained via mutual invocations among services. We show here how an example of a structured cycle can be programmed: while (service c returns true) do {**call** service a}.

```
while :: (ε → Bool) → (ε → T) → Unit
while ⇛ (c)(a) call c >¹ b >
                    IfSignal(b,call a >¹ > call while (c,a)) >¹ x > x
```

where

```
IfSignal(b,P) = if b then P else unit
```

The pattern is programmed as in Orc.

WP17: Interleaved Parallel Routing

"A set of activities is executed in an arbitrary order: each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (*i.e.*, no two activities are active for the same workflow instance at the same time). Example: the Navy requires every job applicant to take two tests: *physical_test* and *mental_test*. These tests can be conducted in any order, but not at the same time."

We assume that each service (a_1 to a_n) signals termination by sending a value, as witnessed by their types. Contrary to Orc, SSCC is expressive enough to describe the pattern within the language.

```
interleave :: [?[!T_1.end]...?[!T_n.end].end]
interleave ⇒ (a_1)...(a_n)(ν back)(
  stream
    back ⇛ (x)feed x
  as lock in
    back ⇐ unit |
    lock(_).a_1 ⇐ (x)(back ⇐ unit) | ... |
    lock(_).a_n ⇐ (x)(back ⇐ unit))
```

Essentially, the different activities execute in parallel, and a lock ensures that two of them are never enabled together. Note the use of auxiliary service back to relay values from the right to the left part of a stream construct, where they are fed into the stream.

In principle, the implementation in SCC can follow the same idea, replacing the stream with a session (possibly obtained using fake services, cf. WP1 above), and the **feed** x with a **return** x. However, the results produced by services a_1 ,..., a_n would naturally go into the same session, and should be forwarded to the top-level. Orc, here, exploits a basic site M implementing a lock with methods acquire and release. This site cannot be programmed inside Orc (see discussion in WP9).

3.2 The automotive scenario

In this section we show how two case studies from an automotive scenario can be represented in SSCC in a satisfactory way. The scenario and the case studies were developed in the EU project Sensoria [40] and the project deliverable D8.0 [23] presents them in detail. The first case study, a sight service that dynamically shows sights according to the driver's preferences, is straightforward to model. The second case study, a dinner service that allows the driver to make a reservation at a restaurant with some interaction, raises some problems with communication and typing of the resulting processes, which we then show that can be solved without compromising the motivation behind SSCC.

Road sights scenario. Here is the scenario as described in [23].

> The driver has subscribed to the dynamic sight service offered by the car company. The vehicle's GPS coordinates are automatically sent to the dynamic sight server at regular intervals, so the vehicle's location is known within a specified radius. Based on the driver's preferences

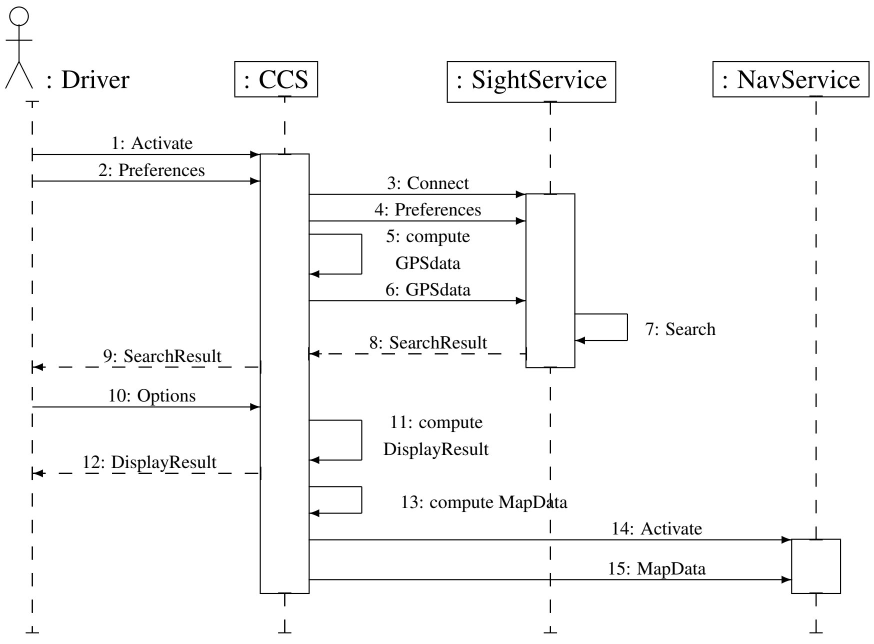

Figure 10: SD for sight service scenario

that were given at the beginning of the trip, the dynamic sight server searches a sightseeing database for appropriate sights and displays them on the in-car map of the vehicle's navigation system. The driver clicks on sights he would like to visit, which results in the display of more detailed information about this specific sight (*e.g.*, opening times, guidance to parking, etc.).

As suggested by Banci et al. [4], there are four actors in this scenario. The driver enables the sight service and sets his preferences via the Car Communication System (CCS). The CCS manages the communication to, and from, the sight service. The sight service has access to a sightseeing database where it gathers information from. Finally, the vehicle's navigation system displays the results to the driver in a graphical way.

UML Sequence Diagrams [3] describe the exchange of messages among the components in a complex system; we use a variant of these diagrams (henceforth referred as SD). The dialogue between the actors is represented in Figure 10. Service CCS can be implemented by the following process in SSCC. The notation ⟨action⟩ stands for an internal action of the system; the numbers on the right correspond to the numbers of the actions in the sequence diagram.

```
CCS ⇒ (Preferences).                          // 1:,2:
      (SightService ⇐ Preferences.            // 3:,4:
                      ⟨compute GPSdata⟩.      // 5:
                      GPSdata.                // 6:
                      (SearchResult).         // 8:
                      feed SearchResult)
      >¹ SearchResult >
      SearchResult.                           // 9:
      (Options).                              // 10:
      ⟨compute DisplayResult⟩.                // 11:
      DisplayResult.                          // 12:
      ⟨compute MapData⟩.                      // 13:
      (NavSystem ⇐ MapData)                   // 14:,15:
```

This implementation follows the SD diagram in Figure 10 closely. It is easy to prove that

$$\Gamma \vdash \mathsf{CCS} \Rightarrow ...: (\textbf{end}, \mathsf{T})$$

for any T (since the process performs no feed to its stream) whenever Γ is such that

$$\begin{aligned}\Gamma &\vdash \mathsf{CCS}: [\mathsf{?Preferences.!SearchResult.?Options.!DisplayResult.}\textbf{end}]\\ \Gamma &\vdash \mathsf{SightService}: [\mathsf{?Preferences.?GPSdata.!SearchResult.}\textbf{end}]\\ \Gamma &\vdash \mathsf{NavSystem}: [\mathsf{?MapData.}\textbf{end}]\end{aligned}$$

The (anonymous) stream has type $\langle\mathsf{SearchResult}\rangle$.

Observe that the types of all these services (CCS, SightService, and NavSystem) correspond precisely to their part of the conversation in the sequence diagram presented above from the point of view of the party who invokes them. For example, the type of SightService composes the arrows labeled 4:, 6:, and 8:, which is the trace of its conversation with CCS.

Instead, the communication along the stream does not correspond to any action in the SD. As we will discuss in more detail in the next section, it corresponds to the communication between two sessions within the CCS (the session CCS–SightService and the session CCS–Driver), and not to communication between different parties.

Dinner service. Once again we present the scenario as described in [23].

> Paul is very hungry, since he is driving without any food for 5 hours, so he activates the dinner service and enters a pizzeria as desired restaurant type and a price range between five and ten euros per meal. The navigation system displays a collection of nearby restaurants that match the preferred settings. Paul chooses the option to check for available seats in the participating local restaurants, and as a result the map displays only restaurants with available tables. Paul chooses "Tony's Pizza" and gets his reservation acknowledged. The way to the restaurant's parking lot is now displayed on the navigation system map.

Again, following the description in Banci et al. [4], we identify four actors in this scenario. The driver enables the dinner service and sets his preferences via the Car Communication System (CCS). The CCS manages the communication to, and from, the dinner service. The dinner service has access to a database of restaurants that it can contact in order to acknowledge the reservation. Finally, the vehicle's navigation system displays the results to the driver in a graphical way.

The dialogue between the actors is represented by the SD in Figure 11, which is again essentially as in [4]. However, it is not so straightforward as before to implement the CCS in SSCC. The problem arises from the need to interact with the driver *after* receiving information from the dinner service, and then give feedback to the latter. We present some alternatives and discuss the drawbacks of each of them.

Implementation using a continuation. In this approach, the dinner service, when invoked for the first time, creates a (customized) new service whose (unique) name is sent back to the CCS. Afterwards, the CCS invokes *that* new service, which contains some persistent information. This solution deviates slightly from the sequence diagram above, since the session between the CCS and the dinner service is actually split into two sessions, one between the CCS and the dinner service (actions 3: to 8:), another between the CCS and the new service (actions 14: and 15:); see Figure 12.

```
CCS ⇒ (Preferences).                                     // 1:,2:
      stream (DinnerService ⇐ Preferences.               // 3:,4:
                              ⟨compute GPSdata⟩.         // 5:
                              GPSdata.                   // 6:
                              (SearchResult).            // 8:
                              feed SearchResult.
                              (NewService).              // 8':
                              feed NewService)
       as f in (f(SearchResult).
                SearchResult.                            // 9:
                (CheckSeats).                            // 10:
                ⟨compute DisplayResult⟩                  // 11:
                DisplayResult.                           // 12:
```

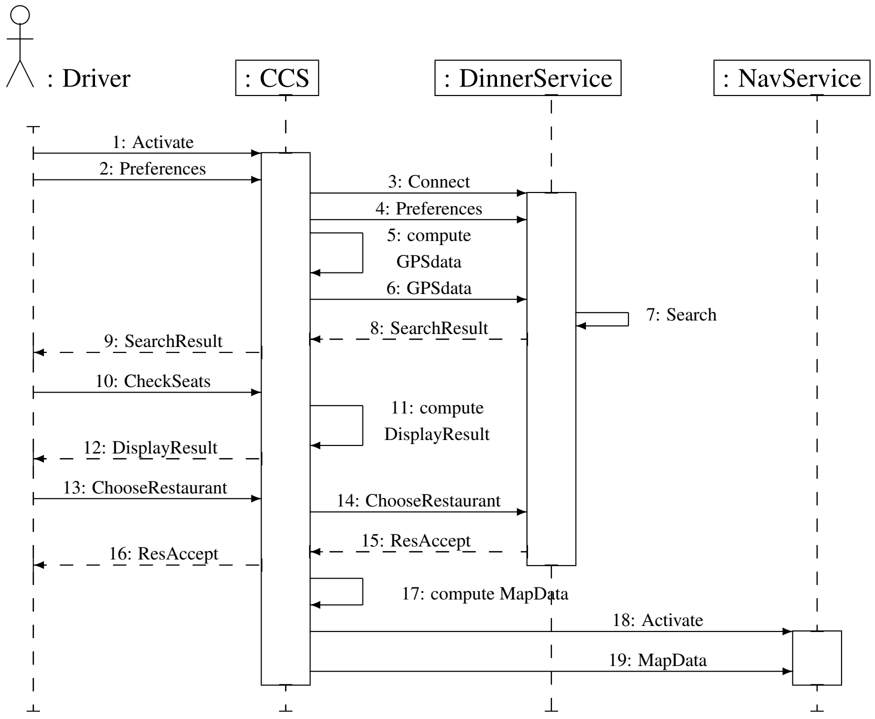

Figure 11: SD for dinner service scenario

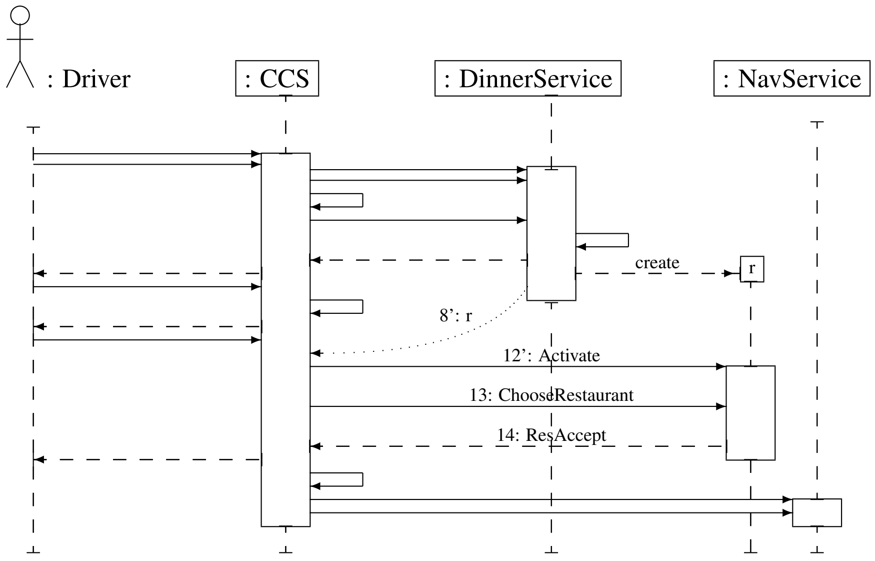

Figure 12: SD for the dinner service scenario with creation of a continuation. Only the continuation part is detailed (cf. Figure 11).

```
                    (ChooseRestaurant).                  // 13:
                    f(NewService).
                    (NewService ⇐ ChooseRestaurant.      // 13':,14:
                                  (ResAccept).           // 15:
                                  feed ResAccept)
                    >¹ ResAccept >
                    ResAccept.                           // 16:
                    ⟨compute MapData⟩.                   // 17:
                    (NavSystem ⇐ MapData)                // 18:,19:
        )
```

This approach is interesting because it shows how continuations can be easily passed as (specialized) services, yielding some form of persistency. However, the CCS as given will not be typable because stream f cannot be typed consistently (since it is used twice to pass two bits of information of different types).

Implementation using an auxiliary service. An alternative approach is to feed new data from the communication system into the dinner service. However, this is not immediately possible using the syntax of SSCC. In order to achieve this "backward" communication, a new (linear) service b is created (whose name is private to the communication system). Whenever the communication system needs to send more data to the dinner service, it does so via b. Notice that one service is created for each message that needs to be sent back.

This solution follows the original SD (Figure 11) faithfully.

```
CCS ⇒ (Preferences).                                          // 1:,2:
      (νb)(stream (DinnerService ⇐ Preferences.               // 3:,4:
                                   ⟨compute GPSdata⟩.         // 5:
                                   GPSdata.                   // 6:
                                   (SearchResult).            // 8:
                                   feed SearchResult.
                                   b ⇓ (ChooseRestaurant).
                                   ChooseRestaurant.          // 14:
                                   (ResAccept).               // 15:
                                   feed ResAccept)
       as f in (f(SearchResult).
                SearchResult.                                 // 9:
                (CheckSeats).                                 // 10:
                ⟨compute DisplayResult⟩.                      // 11:
                DisplayResult.                                // 12:
                (ChooseRestaurant).                           // 13:
                b ⇑ ChooseRestaurant.
                f(ResAccept).
                ResAccept.                                    // 16:
                ⟨compute MapData⟩.                            // 17:
                (NavSystem ⇐ MapData)                         // 18:,19:
      ))
```

where:

- b ⇑ v.P stands for (b ⇐ v.**feed unit**)$>^{1}$ x >P,
  which in turn unfolds to **stream** (b ⇐ v.**feed unit**) **as** g **in** (g(x).P).
- b ⇓ (x).P stands for (b ⇒ (z)**feed** z)$>^{1}$ x >P,
  which in turn unfolds to **stream** (b ⇒ (z)**feed** z) **as** g **in** (g(x).P).

Observe that both b ⇑ v.P and b ⇓ (x).P have the same type as P (but in the second case P might not be typable without the extra information of the type of x); furthermore, v:T ⊢ b:?T, hence the use of these abbreviations always fits well with the type system. Indeed, the only typing problem is the one above —namely stream f cannot be adequately given a type.

Implementation with communication via services. The constructions b $\Uparrow$ v.P and b $\Downarrow$ (x).P may also be used to communicate in the same direction as the stream, hereby avoiding the typing problems both previous solutions suffered from. Thus, we arrive at a third proposal for modeling the dinner service scenario within SSCC, which again follows the proposed SD faithfully.

```
CCS ⇒ (Preferences).                                          // 1:,2:
      (νa₁,a₂,b)(
            (DinnerService ⇐ Preferences.                     // 3:,4:
                             ⟨compute GPSdata⟩.               // 5:
                             GPSdata.                         // 6:
                             (SearchResult).                  // 8:
                             a₁ ⇑ SearchResult.
                             b ⇓ (ChooseRestaurant).
                             ChooseRestaurant.                // 14:
                             (ResAccept).                     // 15:
                             a₂ ⇑ ResAccept)
            |
            (a₁ ⇓ (SearchResult).
             SearchResult.                                    // 9:
             (CheckSeats).                                    // 10:
             ⟨compute DisplayResult⟩.                         // 11:
             DisplayResult.                                   // 12:
             (ChooseRestaurant).                              // 13:
             b ⇑ ChooseRestaurant.
             a₂ ⇓ (ResAccept).
             ResAccept.                                       // 16:
             ⟨compute MapData⟩.                               // 17:
             (NavSystem ⇐ MapData)                            // 18:,19:
            )
      )
```

With this solution, it can be easily verified that

$$\Gamma \vdash \mathsf{CCS} \Rightarrow \ldots : (\textbf{end}, \mathsf{T})$$

for any T (since the process performs no feed to its stream) in any context Γ such that:

$$\begin{aligned}
\Gamma &\vdash \mathsf{CCS}: [?\mathsf{Preferences}.!\mathsf{SearchResult}.?\mathsf{CheckSeats}.\\
&\qquad\qquad !\mathsf{DisplayResult}.?\mathsf{ChooseRestaurant}.!\mathsf{ResAccept}.\textbf{end}]\\
\Gamma &\vdash \mathsf{DinnerService}: [?\mathsf{Preferences}.?\mathsf{GPSdata}.!\mathsf{SearchResult}.\\
&\qquad\qquad ?\mathsf{ChooseRestaurant}.!\mathsf{ResAccept}.\textbf{end}]\\
\Gamma &\vdash \mathsf{NavSystem}: [?\mathsf{MapData}.\textbf{end}].
\end{aligned}$$

The types of services a_1, a_2, and b are derived as in the previous example. Again, the types of the services reflect their communication in the SD with the party who invokes them: the type of CCS is the concatenation of actions 2:, 9:, 10:, 12:, 13:, and 16:; the type of DinnerService is the concatenation of actions 4:, 6:, and 8:; and the type of NavSystem is simply action 19:.

Observe that all internal communication via a_1, a_2, and b is hidden in the type of the services and is not represented in the SD.

Remarks. The Dinner Service scenario exposes an issue with streams: they render communication asymmetric, since a running instance of a service is able to feed information into the process that invoked it, but the latter process has no way to interact back (directly) with that instance of the service (this is actually the intended motivation behind streams, since when a service is invoked it should go on running on its own).

However, in this scenario, the communication system has to synchronize two sessions running concurrently (that with the dinner service and another one with the driver) and information has to run back and forth between them—which is *a priori* not possible due to who invoked whom.

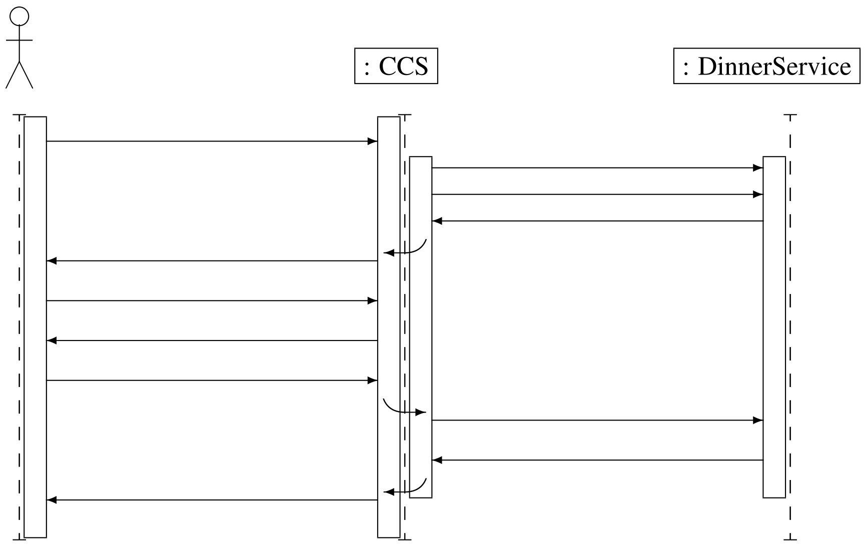

Figure 13: Two services running in parallel (the curved arrows indicate where information needs to be transmitted between sessions)

A simple way to work around this problem is using continuations: whenever a session needs extra information from the context to proceed, it saves its state in a new service, feeds the name of that service into the context and dies; later on, the context can invoke the continuation of that session with any extra information that became available in the meantime. Another possibility is to use ephemerous services to communicate in the "opposite" direction.

Figure 13 depicts the sequence of messages exchanged between the intervenients (forgetting activation commands). Observe that CCS is managing two sessions; the curved arrows denote information that must be transmitted from one into the other (which CCS *should* be able to do). The arrows going left can be implemented by **feed**ing into the appropriate stream; the arrow going right must be dealt with using one of the two mechanisms detailed earlier.

This diagram also explains why typing fails for the first and the second solution. Except for their direction, there is no essential distinction between the arrows connecting both sessions CCS is managing. However, streams are specially tailored to capture the right-to-left arrows, while the left-to-right have to be implemented by service invocation. But the purpose of stream communication is quite different: allowing a process invoking several concurrent services to receive answers from them all regardless the order.

Thus, the search for a symmetric manner of modeling all the curled arrows in Figure 13 is motivated not only because of technical problems, but also by desire for coherence. The third solution proposed to the dinner problem satisfies both requirements: CCS manages communication between the sessions it is involved in in a uniform way, and the resulting process is typable.

This example reveals the need for program transformations that make possible the analysis and verification of programs with such characteristics. The following sections address the behavioral theory of processes that allow us to reason about such program transformations.

4 Behavioral theory

This section deals with the dynamic behavior of SSCC processes. We define first a labeled transition system (LTS) in the early style, and then notions of bisimilarity known in the literature as *strong ground bisimilarity* and *weak ground bisimilarity*. The reason for choosing them is simple: we are interested in capturing contextual equivalence, so bisimilarity should be a congruence. Therefore, we choose the simplest possible setting where this may happen. It is well-known already from the π-calculus that ground bisimilarity over a late LTS is not preserved by parallel composition, requiring the more demanding notions of late and early bisimilarity (which in turn are not preserved by input prefix, since they are not closed under

$\mu \quad ::=$	*Labels*
$\quad \uparrow v$	Value output
$\mid\ \downarrow v$	Value input
$\mid\ a \Rightarrow (r)$	Service definition activation
$\mid\ a \Leftarrow (r)$	Service invocation
$\mid\ \Uparrow v$	Stream feed
$\mid\ f \Downarrow v$	Stream read
$\mid\ r \rhd \uparrow v$	Server session output
$\mid\ r \rhd \downarrow v$	Server session input
$\mid\ r \lhd \uparrow v$	Client session output
$\mid\ r \lhd \downarrow v$	Client session input
$\mid\ r\tau$	Conversation step
$\mid\ \tau$	Internal step
$\mid\ (a) \uparrow a$	Value extrusion
$\mid\ (a) r \rhd \uparrow a$	Server session extrusion
$\mid\ (a) r \lhd \uparrow a$	Client session extrusion
$\mid\ (a) \Uparrow a$	Stream feed extrusion

Figure 14: The syntax of labels

general substitutions). Not surprisingly, this fact also occurs in SSCC: ground bisimilarity is a non-input congruence. Although the general strategy is the same as for π-calculus, the proof techniques themselves differ significantly.

4.1 Labeled transition system

The LTS we define herein adapts (albeit in an insignificant way in terms of expressive power, as discussed below) the original one, presented in [31], to allow for the results we are seeking. There are in total roughly thirty transition rules, which can be grouped into four classes. However, only a few rules correspond to a process actually *doing* something (reading or writing a message); the remaining ones either just propagate an action within a component of a process to the whole system, or model interaction between two separate sub-processes.

Labels and rules. We define first the *actions* performed by a process.

Definition 8 (Labels)**.** *The grammar in Figure 14 defines the syntax of* labels.

We define now the LTS and give a short explanation of the rules.

Definition 9 (Labeled transition system)**.** *The labeled transition system for* SSCC *is inductively defined by the rules in Figure 15.*

The action of sending a value v is represented by the label $\uparrow v$, while receiving v corresponds to the label $\downarrow v$. Feeding a value into a stream is written $\Uparrow v$, reading from stream f is written $f \Downarrow v$. These labels can only arise by application of rules L-SEND, L-RECEIVE, L-FEED, and L-READ.

The request of a service is represented by the label $a \Leftarrow (r)$, and activation by $a \Rightarrow (r)$. These labels arise by means of rules L-CALL and L-DEF. In the labels a is the name of the service and r the fresh name of the session to be created. To ensure freshness, name r is bound in the label. This is obtained thanks to

$$v.P \xrightarrow{\upharpoonleft v} P \qquad (x)P \xrightarrow{\downharpoonleft v} P[{}^{v}\!/\!{}_{x}] \qquad \textsf{\textbf{feed}}\, v.P \xrightarrow{\Uparrow v} P \qquad f(x).P \xrightarrow{f \Downarrow v} P[{}^{v}\!/\!{}_{x}]$$

(L-SEND, L-RECEIVE,L-FEED, L-READ)

$$\frac{r \notin \mathrm{fn}(P)}{a{\Leftarrow}\, P \xrightarrow{a \Leftarrow (r)} r \lhd P} \qquad \frac{r \notin \mathrm{fn}(P)}{a{\Rightarrow}\, P \xrightarrow{a \Rightarrow (r)} r \rhd P}$$

(L-CALL, L-DEF)

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\mu} P'|Q} \qquad \frac{Q \xrightarrow{\mu} Q' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(P) = \emptyset}{P|Q \xrightarrow{\mu} P|Q'}$$

(L-PAR, L-PAR')

$$\frac{P \xrightarrow{\updownarrow v} P'}{r \bowtie P \xrightarrow{r \bowtie \updownarrow v} r \bowtie P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \neq \updownarrow v \quad r \notin \mathrm{bn}(\mu)}{r \bowtie P \xrightarrow{\mu} r \bowtie P'}$$

(L-SESS-VAL, L-SESS-PASS)

$$\frac{P \xrightarrow{\mu} P' \quad \mu \neq \Uparrow v \quad \mathrm{bn}(\mu) \cap (\mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})) = \emptyset}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{\mu} \textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q}$$

(L-STREAM-PASS-P)

$$\frac{Q \xrightarrow{\mu} Q' \quad \mu \neq f \Downarrow v \quad \mathrm{bn}(\mu) \cap (\mathrm{fn}(P) \cup \mathrm{Set}(\vec{w})) = \emptyset}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{\mu} \textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q'}$$

(L-STREAM-PASS-Q)

$$\frac{P \xrightarrow{\Uparrow v} P'}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{\tau} \textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = v :: \vec{w} \,\textsf{\textbf{in}}\, Q}$$

(L-STREAM-FEED)

$$\frac{P \xrightarrow{(v) \Uparrow v} P' \quad v \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{\tau} (\nu\, v)\textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = v :: \vec{w} \,\textsf{\textbf{in}}\, Q}$$

(L-FEED-CLOSE)

$$\frac{Q \xrightarrow{f \Downarrow v} Q'}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} :: v \,\textsf{\textbf{in}}\, Q \xrightarrow{\tau} \textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q'}$$

(L-STREAM-CONS)

$$\frac{P[\textsf{\textbf{rec}}\, X.P/X] \xrightarrow{\mu} P'}{\textsf{\textbf{rec}}\, X.P \xrightarrow{\mu} P'} \qquad \frac{P \xrightarrow{\mu} P' \quad n \notin \mathrm{n}(\mu)}{(\nu\, n)P \xrightarrow{\mu} (\nu\, n)P'}$$

(L-REC, L-RES)

$$\frac{P \xrightarrow{r\tau} P'}{(\nu\, r)P \xrightarrow{\tau} (\nu\, r)P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \in \{\uparrow a, r \bowtie \uparrow a, \Uparrow a\}}{(\nu\, a)P \xrightarrow{(a)\mu} P'}$$

(L-SESS-RES, L-EXTR)

$$\frac{P \xrightarrow{a \Leftrightarrow (r)} P' \quad Q \xrightarrow{a\, \overline{\Leftrightarrow} (r)} Q'}{P|Q \xrightarrow{\tau} (\nu\, r)(P'|Q')} \qquad \frac{P \xrightarrow{a \Leftrightarrow (r)} P' \quad Q \xrightarrow{a\, \overline{\Leftrightarrow} (r)} Q'}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{\tau} (\nu\, r)\textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q'}$$

(L-SERV-COM-PAR, L-SERV-COM-STREAM)

$$\frac{P \xrightarrow{r \bowtie \upharpoonleft v} P' \quad Q \xrightarrow{r\, \overline{\bowtie} \downharpoonleft v} Q'}{P|Q \xrightarrow{r\tau} P'|Q'} \qquad \frac{P \xrightarrow{r \bowtie \upharpoonleft v} P' \quad Q \xrightarrow{r\, \overline{\bowtie} \downharpoonleft v} Q'}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{r\tau} \textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q'}$$

(L-SESS-COM-PAR, L-SESS-COM-STREAM)

$$\frac{P \xrightarrow{r \bowtie (v) \upharpoonleft v} P' \quad Q \xrightarrow{r\, \overline{\bowtie} \downharpoonleft v} Q' \quad v \notin \mathrm{fn}(Q)}{P|Q \xrightarrow{r\tau} (\nu\, v)(P'|Q')} \qquad \frac{P \xrightarrow{r \bowtie \downharpoonleft v} P' \quad Q \xrightarrow{r\, \overline{\bowtie} (v) \upharpoonleft v} Q' \quad v \notin \mathrm{fn}(P)}{P|Q \xrightarrow{r\tau} (\nu\, v)(P'|Q')}$$

(L-PAR-CLOSE, L-PAR-CLOSE')

$$\frac{P \xrightarrow{r \bowtie (v) \upharpoonleft v} P' \quad Q \xrightarrow{r\, \overline{\bowtie} \downharpoonleft v} Q' \quad v \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{r\tau} (\nu\, v)\textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q'}$$

(L-SESS-COM-CLOSE)

$$\frac{P \xrightarrow{r \bowtie \downharpoonleft v} P' \quad Q \xrightarrow{r\, \overline{\bowtie} (v) \upharpoonleft v} Q' \quad v \notin \mathrm{fn}(P) \cup \mathrm{Set}(\vec{w})}{\textsf{\textbf{stream}}\, P \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q \xrightarrow{r\tau} (\nu\, v)\textsf{\textbf{stream}}\, P' \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, Q'}$$

(L-SESS-COM-CLOSE')

Figure 15: The labeled transition system

side conditions of propagation rules below, which guarantee that bound names in the labels are different from free names in the process.

A third group of rules is concerned with propagation of labels, defining the circumstances when a process executes an action because one of its components executes that action. Labels propagate across parallel composition (rules L-PAR and L-PAR') or session endpoints (as long as they do not bind the session names); in this case, the labels for message sending/receiving become prefixed with the session name (rule L-SESS-VAL), while other labels remain unchanged (rule L-SESS-PASS). Here, $\updownarrow v$ stands for one of $\downarrow v$ or $\uparrow v$, and multiple occurrences of $\updownarrow v$ are assumed to be instantiated in the same way, and likewise $a \Leftrightarrow (r)$ stands for one of $a \Rightarrow (r)$ and $a \Leftarrow (r)$.

Every label is also propagated across streams (rules L-STREAM-PASS-P and L-STREAM-PASS-Q) except for feeding or reading by the process at the stream's appropriate end. In the latter cases, the value is fed into/read from the stream, and the process executes an internal action τ (rules L-STREAM-FEED and L-STREAM-CONS); if the value fed is restricted (see below), rule L-FEED-CLOSE should be applied instead. Recursion and restriction do not affect labels (rules L-REC and L-RES), in the latter case as long as these do not refer to the name being restricted. The exception occurs when an action with label $r\tau$ (see below) occurs within a process where r is restricted: it becomes simply an internal action τ. In the remaining cases, name extrusion occurs (rule L-EXTR). Name extrusion may occur whenever a value is communicated, i.e. in value sending (this also includes labels prefixed by a session name) and in value feed. As a notation, the bound name is written in parentheses.

Internal actions, labeled τ, and actions internal to session r, labeled $r\tau$, arise when two subprocesses interact. This can happen when a service invocation meets a service activation (rules L-SERV-COM-PAR and L-SERV-COM-STREAM) or when two endpoints of a session r communicate (rules L-SESS-COM-PAR and L-SERV-COM-PAR); here name extrusion may occur, in which case rules L-PAR-CLOSE or L-SESS-COM-CLOSE apply. In these rules, $\overline{\updownarrow} v$ stands for the opposite instantiation of $\updownarrow v$ and $a \overline{\Leftrightarrow} (r)$ for the opposite instantiation of $a \Leftrightarrow (r)$.

A few comments are in place at this point. The original LTS for SSCC, presented in [31], had fewer rules (in particular, all the propagation rules were absent) and included the following additional rule.

$$\frac{P \xrightarrow{\mu} P' \quad P \equiv Q \quad P' \equiv Q'}{Q \xrightarrow{\mu} Q'} \qquad \text{(L-STRUCT)}$$

Unfortunately, allowing this rule removes the capacity to use structural induction over processes to reason over their actions. Furthermore, although all the new rules are admissible in the original presentation of SSCC, rule L-STRUCT is not admissible in the new calculus, so the latter is strictly weaker. To see why, note that in the original LTS, using L-STRUCT, L-SEND and L-PAR, one can infer that $a|b \xrightarrow{\uparrow a} b$. However, without L-STRUCT, the best one can show is that $a|b \xrightarrow{\uparrow a} \mathbf{0}|b$. Notice that, in the example, one still has $\mathbf{0}|b \equiv b$, which for practical purposes suffices. The following result shows that this is not a coincidence.

Lemma 1 (Harmony Lemma). *Let P and Q be processes with $P \equiv Q$. If $P \xrightarrow{\alpha} P'$, then $Q \xrightarrow{\alpha} Q'$ for some Q' with $P' \equiv Q'$, and vice-versa.*

The proof of this result is in Appendix C, together with lemmas showing the derivability of the new transition rules in the original system. The new LTS is strictly weaker than the original one, since it is not always the case that two structurally congruent processes can evolve via the same action to the same process, as the example above shows. However, it is always the case that structurally congruent processes can evolve via the same action to structurally congruent processes: this is precisely the statement of the Harmony Lemma. Observe that this is sufficient for both LTSs to yield the same notion of bisimilarity: the Harmony Lemma implies that structural equivalence is a bisimulation, while all other transition rules are preserved.

The reduction relation on processes (Definition 5) coincides with the restriction of the LTS to τ-transitions.

Lemma 2 (Correspondence Lemma). *Let P and Q be processes. It is the case that $P \xrightarrow{\tau} Q$ if, and only if, $P \to Q$.*

4.2 Bisimilarity notions

The first notion of equivalence between processes that we present is strong bisimilarity, hereafter referred to simply as "bisimilarity". Two bisimilar processes behave in the same way, in the sense that they mimic each other's actions (even those that are not visible to the outside) perfectly.

Definition 10.

- *A symmetric binary relation $\mathcal{R}$ on processes is a* (strong) bisimulation *if, for any processes P, Q such that $P \mathcal{R} Q$, if $P \xrightarrow{\alpha} P'$ for some process P' and action α such that no bound name in α is free in P or Q, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ with $P' \mathcal{R} Q'$.*
- *(Strong) bisimilarity $\sim$ is the largest bisimulation.*
- *Two processes P and Q are said to be* (strongly) bisimilar *if $P \sim Q$.*

Moreover, a full *bisimulation is a bisimulation closed under service name substitutions, and we call full bisimilarity $\sim_{\mathrm{f}}$ the largest full bisimulation.*

Since bisimilarity is not closed under service name substitutions, $\sim_{\mathrm{f}} \subsetneq \sim$. Bisimilarity (respectively full bisimilarity) can be obtained both as the union of all bisimulations (respectively full bisimulations) or as a fixed-point of a suitable monotonic operator; both characterizations are useful. Notice that "equal" (structurally congruent) processes are bisimilar, since by the Harmony Lemma, $\equiv$ is a bisimulation. Moreover, structurally congruent processes are also fully bisimilar. Bisimilarity as just defined is a non-input congruence, as it is the case in the π-calculus.

Definition 11.

- *A* context *$C[\![]\!]$ is a process where exactly one occurrence of* $\mathbf{0}$ *has been replaced by a hole $\bullet$. Given a process P, $C[\![P]\!]$ is the process obtained by replacing the hole in $C[\![]\!]$ by P.*
- *A context is said to be* non-input *if no hole occurs under an input prefix (x) or $f(x)$.*

Note that here we consider all possible contexts, not only active contexts as defined in Definition 4.

Proposition 1. *Bisimilarity is a non-input congruence: if $P \sim Q$ and $C[\![]\!]$ is a non-input context, then $C[\![P]\!] \sim C[\![Q]\!]$.*

The proof is detailed in Appendix D. The strategy is the same as in [41], based on the notion and properties of a relation *progressing* to another relation; however, several major details of the proof are different due to the presence of recursion in the syntax of processes.

Corollary 1. *Full bisimilarity is a congruence.*

When discussing the behavior of a system, one often wants to abstract from implementation details and ignore internal actions. In particular, processes are deemed to be equivalent if their *visible* behaviors coincide. This is the goal of weak bisimilarity. Write $P \overset{\tau}{\Longrightarrow} Q$ whenever $P \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q$, and $P \overset{\alpha}{\Longrightarrow} Q$ whenever $P \overset{\tau}{\Longrightarrow}\xrightarrow{\alpha}\overset{\tau}{\Longrightarrow} Q$ for $\alpha \neq \tau$. Notice that, in particular, $P \overset{\tau}{\Longrightarrow} P$ for every process P.

Definition 12.

- *A symmetric binary relation $\mathcal{R}$ on processes is a* weak bisimulation *if, for any processes P, Q such that $P \mathcal{R} Q$, if $P \overset{\alpha}{\Longrightarrow} P'$ for some process P' and action α such that no bound name in α is free in P or Q, there exists a process Q' such that $Q \overset{\alpha}{\Longrightarrow} Q'$ with $P' \mathcal{R} Q'$.*
- *Weak bisimilarity $\approx$ is the largest weak bisimulation.*
- *Two processes P and Q are said to be* weakly bisimilar *if $P \approx Q$.*

Moreover, a full *weak bisimulation is a weak bisimulation closed under service name substitutions, and we call full weak bisimilarity $\approx_{\mathrm{f}}$ the largest full weak bisimulation.*

Again, (full) weak bisimilarity can be obtained as the union of all (full) weak bisimulations or as a fixed-point of a suitable monotonic operator. The main property of weak bisimilarity is, as before, the following.

Proposition 2. *Weak bisimilarity is a non-input congruence.*

The proof is detailed in Appendix D. Note that the typical π-calculus examples showing that (weak) bisimulation is not a congruence can be adapted in a straightforward way to SSCC.

Corollary 2. *Full weak bisimilarity is a congruence.*

4.3 Useful axioms

Even if presenting a complete axiomatization for a calculus as complex as SSCC is out of the scope of this paper, it is interesting to present some axioms (equational laws correct with respect to full strong bisimilarity) that capture key facts about the behavior of processes. Some of them will also prove to be useful in the next session.

On streams and sessions. In all cases but Axiom 8 (whose proof is described below), the correctness of each axiom below is straightforwardly proved by considering a full bisimulation including all the instances of the axiom together with the identity. The notation $\{\textbf{feed}\, v.Q \to Q\}$ in the Unused Stream law (Axiom 7) denotes a transformation on processes, defined by induction on the syntax, that is an homomorphism for all process constructors, except for **feed**, where it transforms $\textbf{feed}\, v.Q$ into Q, for all v and Q.

Proposition 3.

Session Garbage Collection

$$(\nu\, \mathsf{r})\mathcal{D}[\![\mathsf{r} \rhd \mathbf{0}, \mathsf{r} \lhd \mathbf{0}]\!] \sim_{\mathrm{f}} \mathcal{D}[\![\mathbf{0}, \mathbf{0}]\!] \qquad \textit{if } \mathcal{D} \textit{ does not bind } r \tag{1}$$

Stream Garbage Collection

$$\textbf{stream}\, 0\, \textbf{as}\, f\, \textbf{in}\, P \sim_{\mathrm{f}} P \qquad \textit{if } f \textit{ does not occur in } P \tag{2}$$

Session Independence

$$r \bowtie \mathsf{Q} \,|\, s \bowtie \mathsf{P} \sim_{\mathrm{f}} r \bowtie (s \bowtie \mathsf{Q} \,|\, \mathsf{P}) \qquad \textit{if } s \neq r \tag{3}$$

The same holds if the sessions have opposite polarities:

$$r \bowtie \mathsf{Q} \,|\, s \overline{\bowtie} \mathsf{P} \sim_{\mathrm{f}} r \bowtie (s \overline{\bowtie} \mathsf{Q} \,|\, \mathsf{P}) \qquad \textit{if } s \neq r$$

Stream Independence

$$\begin{aligned}&\textbf{stream}\, P\, \textbf{as}\, f\, \textbf{in}\, \textbf{stream}\, P'\, \textbf{as}\, g\, \textbf{in}\, Q \sim_{\mathrm{f}} \\ &\qquad \textbf{stream}\, P'\, \textbf{as}\, g\, \textbf{in}\, \textbf{stream}\, P\, \textbf{as}\, f\, \textbf{in}\, Q \qquad \textit{if } f \neq g\end{aligned} \tag{4}$$

Streams are Orthogonal to Sessions

$$r \bowtie (\textbf{feed}\, v \,|\, P) \sim_{\mathrm{f}} \textbf{feed}\, v \,|\, r \bowtie P \tag{5}$$

Stream Locality

$$\textbf{stream}\, P\, \textbf{as}\, f\, \textbf{in}\, (Q \,|\, Q') \sim_{\mathrm{f}} (\textbf{stream}\, P\, \textbf{as}\, f\, \textbf{in}\, Q) \,|\, Q' \qquad \textit{if } f \notin \mathrm{fn}(Q') \tag{6}$$

Unused Stream

$$\textbf{stream}\, P\, \textbf{as}\, f\, \textbf{in}\, 0 \approx_{\mathrm{f}} P\{\textbf{feed}\, v.Q \to Q\} \tag{7}$$

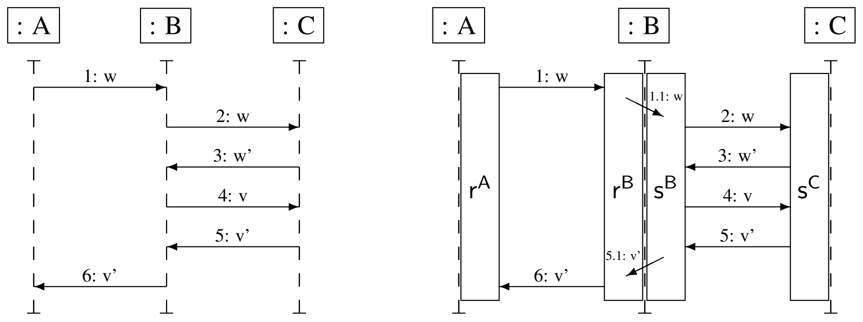

Figure 16: SD for communication pattern: object-centred and session-centred view

Parallel Composition Versus Streams

$$\textbf{stream}\ P\ \textbf{as}\ f\ \textbf{in}\ Q \sim_{\mathrm{f}} P|Q \quad \textit{if } f \notin \mathrm{fn}(Q) \textit{ and } P \textit{ does not contain } \textbf{feed} \tag{8}$$

The Session Independence law shows that different sessions are independent. Interestingly this property is strongly connected to the operators available in SSCC, failing in similar calculi such as SCC [8]. Note that Axiom 7 is correct only with respect to full weak bisimilarity. It becomes correct with respect to full strong bisimilarity if and only if P does not contain any feed which is not inside another stream. We still have to prove Axiom 8.

Soundness of Equation 8. Since $Q \equiv \mathbf{0}|Q$ we can apply Axiom 6 to obtain

$$\textbf{stream}\ P\ \textbf{as}\ f\ \textbf{in}\ Q \sim_{\mathrm{f}} (\textbf{stream}\ P\ \textbf{as}\ f\ \textbf{in}\ \mathbf{0})|Q$$

The thesis then follows from Axiom 7. □

5 Program transformations

This section presents an application of the results discussed earlier by means of a more complex example. The first subsection discusses different approaches to implement services as SSCC processes; then a sequence of transformation rules is presented mapping processes obtained by one approach into processes obtained by the other. The results in Section 4, together with general results to break sequential sessions into smaller sessions developed in Section 5.3, are used to establish soundness of the transformation.

5.1 Program design

We describe how the same behavior, initially specified in an object-oriented style, can be modeled first in a more session-centered style, and then in a request-response style suitable for implementation. Diagrams for the initial sequence diagram and those describing the result of each transformation are in Figures 16 to 18.

The object-oriented view. The SD on the left of Figure 16 describes a very common pattern appearing in scenarios involving (at least) three partners. The description of the communication pattern is as follows.

> Object B receives from object A the value w and forwards it to object C. After receiving the value, object C answers with a value w'. Object B replies with v and, finally, object C replies with v'. Now object B forwards it to object A.

Notice that by "Object B receives from object A the value w" we mean that object A invokes a method in object B passing the value w.

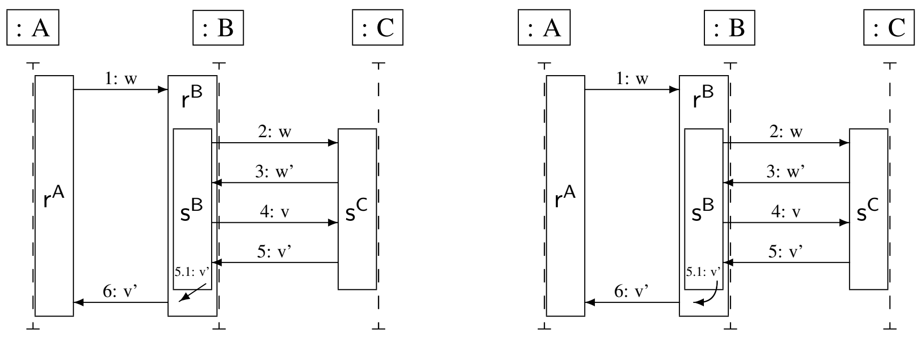

Figure 17: SD for communication pattern: using a subsession

The session-centered view. Assume that the components of this abstract communication scenario are clients and servers of a service-oriented architecture, and further assume that communication happens via sessions. We refine the diagram by incorporating information about the running sessions, in the diagram on the right of Figure 16, where the slanted arrows mean message passing between sessions[4]. An instance of service B (let us call *participant* such an instance) has a session r running with participant A and another session s running with participant C. Since sessions involve two partners, a session r between participants A and B has two sides—called endpoints, r^A at participant A and r^B at participant B. The communication pattern is now like this:

> Participant B receives in session r^B the value w, passes it to its part of the session with participant C (s^B), and then forwards the value through this session to C. Inside the same session, C sends w' to B, B sends v to C and C sends v' to B. Participant B now forwards the value v' back to A, passing it from session s to session r.

In addition to the normal constructs in the calculus, to model object-oriented systems (that do not follow the laws of session communication), it is useful to have two constructs enabling arbitrary message passing. These can be obtained by generalizing auxiliary services to polyadic communications:

$$b \Uparrow \langle v_1, ..., v_n \rangle.P \triangleq \textsf{\textbf{stream}}\, b \Leftarrow v_1 ... v_n.\textsf{\textbf{feed unit as}}\ f\ \textsf{\textbf{in}}\ f(v).P$$

$$b \Downarrow (x_1, ..., x_n)P \triangleq \textsf{\textbf{stream}}\, b \Rightarrow (z_1)...(z_n).\textsf{\textbf{feed}}\ z_1 ... \textsf{\textbf{feed}}\ z_n\ \textsf{\textbf{as}}\ f$$
$$\textsf{\textbf{in}}\ f(x_1)...f(x_n).P$$

where name v and stream f are not used in P.

The diagram on the right of Figure 16 is directly implemented in SSCC as

$$\mathsf{SC} \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{B} \mid \mathsf{C}\ ),$$

where

$$\mathsf{A} \triangleq b \Leftarrow w.(y)P,\ \mathsf{B} \triangleq (\nu\, b_1, b_2)\ (\ \mathsf{B}_1 \mid \mathsf{B}_2\ ),\ \text{and}\ \mathsf{C} \triangleq c \Rightarrow (x)w'.(y)v'.S,$$

with

$$\mathsf{B}_1 \triangleq b \Rightarrow (x)b_1 \Uparrow x.b_2 \Downarrow (y)y.Q,\ \text{and}\ \mathsf{B}_2 \triangleq c \Leftarrow b_1 \Downarrow (x)x.(z)v.(y)b_2 \Uparrow y.R.$$

The process above, although not deterministic (*e.g.*, its first step may either be the invocation of service b or of service c), is confluent, and it is easy to check that its behavior reflects the one described on the right of Figure 16.

[4]Since to the best of our knowledge no extension of Sequence Diagrams with session information exists, we introduce in SD a notation for it.

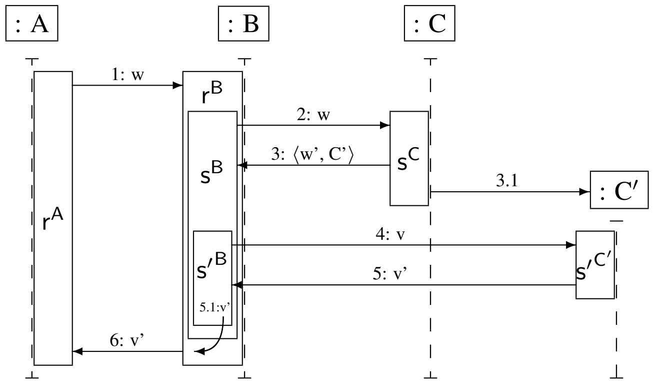

Figure 18: SD for communication pattern: using subsessions and continuations

A first optimization. When participant B receives the value sent by A, it may immediately send it to participant C, by calling it (and thus opening a subsession). One simply has to perform a "local" transformation on B. The resulting diagram is on the left of Figure 17, and it is implemented in SSCC as process SC′, where we denote by E the new instance of B.

$$\mathsf{SC}' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{E} \mid \mathsf{C}\ )$$
$$\mathsf{E} \triangleq b \Rightarrow (x)(\nu\, b_1)(c \Leftarrow x.(z)v.(y)b_1 \Uparrow y.R \mid b_1 \Downarrow (y)y.Q)$$

Still, participant E (which replaces B from the previous version) needs to pass the value sent by C in their session, to its session with A, and a communication based on an auxiliary service is used.

A second optimization. The transfer of a value from a subsession to its parent session is, in the previous implementation, not straightforward, since it requires the use of local services. In fact, it is more convenient to use a trans-session construct like **stream**. Now participant F (initially B), passes the value received from C, from the subsession to the main session, using another communication construct—a stream—instead of using a local service. This is implemented in process SC″.

$$\mathsf{SC}'' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{F} \mid \mathsf{C}\ )$$

where the new code for B (now F) is as follows.

$$\mathsf{F} \triangleq b \Rightarrow (x)(\textbf{stream}\ c \Leftarrow x.(z)v.(y)\textbf{feed}\, y.R\ \textbf{as}\ f\ \textbf{in}\ f(y).y.Q)$$

The corresponding diagram is on the right of Figure 17. The two diagrams in Figure 17 are quite similar: the one on the left uses message passing (denoted by the straight arrow from the inner to the outer session), whereas the one on the right uses stream communication (described by the curved arrow).

Implementing the diagram. The previous diagram, and the corresponding SSCC process, model the pattern at hand in a service-oriented, session-based style. However, current Web Service technologies do not provide support for a complex mechanism such as sessions, considering instead request (one-way communication) and request-response (two one-way communications in opposite directions) only—see, *e.g.*, the definition of WSDL [18]. It is easy to see that these are particular cases of sessions, where protocols are composed respectively by one output or by one output followed by one input. The new communication pattern is described in Figure 18, and reads as follows.

> Participant B receives in session r^B the value w, and then forwards it through its session s^B with participant C to C itself. Inside the same session C sends to B value w′ together with the name of a freshly generated service C′ to continue the conversation on. Now B invokes C′ creating a new subsession s′ of session s and, inside s′, B sends v and receives as answer v′. Participant B now forwards the value v′ back to A, passing it from session s′ to session r.

This pattern can be implemented as:

$$\mathsf{SC}''' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{G} \mid \mathsf{D}\ )$$

where the new codes for B (now G) and C (now D) are below. To write these new codes we need to consider polyadic inputs and outputs, denoted respectively by $(x_1, \ldots, x_n)$ and $\langle v_1, \ldots, v_n \rangle$. They can be easily accommodated in the theory.

$$\mathsf{G} \triangleq b \Rightarrow (x)(\textbf{stream } c \Leftarrow x.(z, c')c' \Leftarrow v.(y)\textbf{feed } y.R \textbf{ as } f \textbf{ in } f(y).y.Q)$$
$$\mathsf{D} \triangleq c \Rightarrow (x)(\nu\, c')\langle w', c'\rangle.c' \Rightarrow (y)v'.S$$

Naturally, one asks whether the transformations of SC into SC$'$, SC$''$, and, finally, SC$'''$ are correct, not changing the observable behavior of processes. Next we show how to give a positive answer to this question.

5.2 Soundness of the transformations

We now prove that the transformations presented are actually correct with respect to full weak bisimilarity. Interestingly, they are also transparent for process A, *i.e.*, A needs not to be changed when the transformation is applied. To prove this we show that the three equations below hold.

$$(\nu\, \mathsf{c})(\mathsf{B} \mid \mathsf{C}) \approx_{\mathrm{f}} (\nu\, \mathsf{c})(\mathsf{E} \mid \mathsf{C}) \tag{9}$$
$$(\nu\, \mathsf{c})(\mathsf{E} \mid \mathsf{C}) \approx_{\mathrm{f}} (\nu\, \mathsf{c})(\mathsf{F} \mid \mathsf{C}) \tag{10}$$
$$(\nu\, \mathsf{c})(\mathsf{F} \mid \mathsf{C}) \approx_{\mathrm{f}} (\nu\, \mathsf{c})(\mathsf{G} \mid \mathsf{D}) \tag{11}$$

The correctness of the whole transformations, *i.e.*, SC $\approx_{\mathrm{f}}$ SC$'$ $\approx_{\mathrm{f}}$ SC$''$ $\approx_{\mathrm{f}}$ SC$'''$ follows by closing under contexts the equations above.

Note that the transformation of auxiliary communications (passing value w from r^{B} to s^{B} and value v from s^{B} to r^{B}) into normal communications are not correct with respect to full strong bisimilarity. In fact, auxiliary communications require a few more steps, and leave behind them empty sessions and streams, which have to be garbage collected. The correctness of garbage collection is based on Axioms 1 and 2 (cfr. Section 4.3).

Soundness of Equation 9. The proof can be easily obtained by exhibiting a bisimulation including the two processes. For simplicity we will not detail it, but just highlight a few important points. For easier reading we recall B, E and C below.

$$\mathsf{B} \triangleq (\nu\, b_1, b_2)\ (\ b \Rightarrow (x) b_1 \Uparrow x.b_2 \Downarrow (y) y.Q \mid c \Leftarrow b_1 \Downarrow (x) x.(z) v.(y) b_2 \Uparrow y.R\ )$$
$$\mathsf{E} \triangleq b \Rightarrow (x)(\nu\, b_1)(c \Leftarrow x.(z)v.(y)b_1 \Uparrow y.R \ \mid\ b_1 \Downarrow (y)y.Q)$$
$$\mathsf{C} \triangleq c \Rightarrow (x)w'.(y)v'.S$$

The two processes can mimic each other even if they are nondeterministic, since the nondeterminism comes from τ steps, whose order is not important, given that processes are confluent. We thus just consider a possible order of execution of τ steps. Both the processes can execute the following sequence of actions: a service invocation at b giving rise to session r, an input on x taking value w, a service invocation at c giving rise to session s. Then B can execute the auxiliary communication along b_1. Then both B and C can execute their protocol with C. By executing the two missing auxiliary communications $(\nu\, \mathsf{c})(\mathsf{B} \mid \mathsf{C})$ and $(\nu\, \mathsf{c})(\mathsf{E} \mid \mathsf{C})$ reduce respectively to:

$$(\nu\, \mathsf{s})(r \rhd \mathsf{Q}[{}^{w}\!/\!{}_{x}][{}^{v'}\!/\!{}_{y}] \mid s \lhd \mathsf{R}[{}^{w}\!/\!{}_{x}][{}^{w'}\!/\!{}_{z}][{}^{v'}\!/\!{}_{y}] \mid s \rhd \mathsf{S}[{}^{w}\!/\!{}_{x}][{}^{v}\!/\!{}_{y}])$$
$$(\nu\, \mathsf{s})(r \rhd (s \lhd \mathsf{R}[{}^{w}\!/\!{}_{x}][{}^{w'}\!/\!{}_{z}][{}^{v'}\!/\!{}_{y}]) \mid \mathsf{Q}[{}^{w}\!/\!{}_{x}][{}^{v'}\!/\!{}_{y}] \mid s \rhd \mathsf{S}[{}^{w}\!/\!{}_{x}][{}^{v}\!/\!{}_{y}])$$

where we used structural congruence (which is included in full bisimilarity, according to Theorem 6 in Appendix D), garbage collection Axioms 1 and 2, and closure under contexts to remove the garbage produced by auxiliary communications.

The processes above can be proved equivalent using structural congruence, session independence (Axiom 3) and closure under contexts. □

Soundness of Equation 10. To prove the correctness of Equation 10 it is enough to prove $E \approx_{\mathrm{f}} F$, then the thesis follows from closure under contexts. For easier reading we recall E and F below.

$$\mathsf{E} \triangleq b \Rightarrow (x)(\nu\, b_1)(c \Leftarrow x.(z)v.(y)b_1 \Uparrow y.R \mid b_1 \Downarrow (y)y.Q)$$
$$\mathsf{F} \triangleq b \Rightarrow (x)(\textbf{stream}\ c \Leftarrow x.(z)v.(y)\textbf{feed}\ y.R\ \textbf{as}\ f\ \textbf{in}\ f(y).y.Q)$$

Actually, in general we can prove

$$(\nu\, \mathsf{a})(\mathcal{C}'[\![a \Uparrow v.P]\!] \mid \mathcal{C}''[\![a \Downarrow (y).Q]\!]) \approx_{\mathrm{f}} \textbf{stream}\ \mathcal{C}'[\![\textbf{feed}\ v.P]\!]\ \textbf{as}\ f\ \textbf{in}\ \mathcal{C}''[\![f(y).Q]\!] \qquad (12)$$

provided that neither a nor f occur elsewhere and P and $\mathcal{C}'$ contain no feeds. The thesis will follow by instantiation and closure under contexts.

The proof shows that the three pairs below, together with a few other pairs differing from these because of τ transitions (corresponding to intermediate steps), form a full bisimulation.

$$((\nu\, \mathsf{a})(\mathcal{C}'[\![a \Uparrow v.P]\!] \mid \mathcal{C}''[\![a \Downarrow (y).Q]\!]), \textbf{stream}\ \mathcal{C}'[\![\textbf{feed}\ v.P]\!]\ \textbf{as}\ f\ \textbf{in}\ \mathcal{C}''[\![f(y).Q]\!])$$
$$(\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\textbf{stream}\ \mathbf{0}\ \textbf{as}\ f' = \langle v \rangle\ \textbf{in}\ f'(y).Q]\!],$$
$$\textbf{stream}\ \mathcal{C}'[\![P]\!]\ \textbf{as}\ f = \langle v \rangle\ \textbf{in}\ \mathcal{C}''[\![f(y).Q]\!])$$
$$(\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![Q]\!], \textbf{stream}\ \mathcal{C}'[\![P]\!]\ \textbf{as}\ f\ \textbf{in}\ \mathcal{C}''[\![Q]\!])$$

In each process considered in the relation, a, f, and f' do not occur elsewhere, and P and $\mathcal{C}'$ do not contain feeds. The only difficult part is when on the right the feed and the read from stream are executed and, correspondingly, the two auxiliary communications on the left. Actually, both transitions on the right amount to τ actions.

$$\textbf{stream}\ \mathcal{C}'[\![\textbf{feed}\ v.P]\!]\ \textbf{as}\ f\ \textbf{in}\ \mathcal{C}''[\![f(y).Q]\!] \xrightarrow{\tau} \textbf{stream}\ \mathcal{C}'[\![P]\!]\ \textbf{as}\ f = \langle v \rangle\ \textbf{in}\ \mathcal{C}''[\![f(y).Q]\!]$$

$$\textbf{stream}\ \mathcal{C}'[\![P]\!]\ \textbf{as}\ f = \langle v \rangle\ \textbf{in}\ \mathcal{C}''[\![f(y).Q]\!] \xrightarrow{\tau} \textbf{stream}\ \mathcal{C}'[\![P]\!]\ \textbf{as}\ f\ \textbf{in}\ \mathcal{C}''[\![Q[{}^{v}\!/\!_{y}]]\!]$$

The first transition is matched as follows.

$$(\nu\, \mathsf{a})\mathcal{C}'[\![a \Uparrow v.P]\!] \mid \mathcal{C}''[\![a \Downarrow (y).Q]\!] =$$
$$(\nu\, \mathsf{a})\mathcal{C}'[\![\textbf{stream}\ a \Leftarrow v.\textbf{feed}\ \textbf{unit}\ \textbf{as}\ f''\ \textbf{in}\ f''(x).P]\!] \mid$$
$$\mathcal{C}''[\![\textbf{stream}\ a \Rightarrow (z)\textbf{feed}\ z\ \textbf{as}\ f'\ \textbf{in}\ f'(y).Q]\!] \xrightarrow{\tau}{}^{*}$$
$$(\nu\, \mathsf{a}, \mathsf{r})\mathcal{C}'[\![\textbf{stream}\ r \lhd \mathbf{0}\ \textbf{as}\ f''\ \textbf{in}\ P]\!] \mid \mathcal{C}''[\![\textbf{stream}\ r \rhd \mathbf{0}\ \textbf{as}\ f' = \langle v \rangle\ \textbf{in}\ f'(y).Q]\!] \sim_{\mathrm{f}}$$
$$\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\textbf{stream}\ \mathbf{0}\ \textbf{as}\ f' = \langle v \rangle\ \textbf{in}\ f'(y).Q]\!]$$

In the first step we used the definitions of auxiliary communications. The sequence of τ actions includes service invocation at a creating session r, communication of v along the session, the two feeds and the read from stream f''. In the last step we used Axioms 1 and 2 for garbage collection. For the challenge from the left, note that only τ actions are involved, thus they can be matched by the right term staying idle, and the order in which they are executed is not relevant.

The second transition from the right is matched by:

$$\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\textbf{stream}\ \mathbf{0}\ \textbf{as}\ f' = \langle v \rangle\ \textbf{in}\ f'(y).Q]\!] \xrightarrow{\tau}$$
$$\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\textbf{stream}\ \mathbf{0}\ \textbf{as}\ f'\ \textbf{in}\ Q[{}^{v}\!/\!_{y}]]\!] \sim_{\mathrm{f}} \mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![Q[{}^{v}\!/\!_{y}]]\!]$$

where we used Axiom 2 again. This concludes the proof. □

Equation 11 only holds when processes are sequential. Therefore, we adapt the type system for that purpose and present then the envisaged result.

$$\frac{P\colon U}{v.P\colon !.U} \quad \frac{P\colon U}{(x)P\colon ?.U} \quad \frac{P\colon U}{a \Rightarrow P\colon \textsf{end}} \quad \frac{P\colon U}{a \Leftarrow P\colon \textsf{end}} \qquad \text{(T-SEND, T-RECEIVE, T-DEF, T-CALL)}$$

$$\frac{P\colon U}{r \rhd P\colon \textsf{end}} \quad \frac{P\colon U}{r \lhd P\colon \textsf{end}} \quad \frac{P\colon U}{\textsf{feed}\, v.P\colon U} \quad \frac{P\colon U}{f(x).P\colon U} \qquad \text{(T-SESS-S, T-SESS-C, T-FEED, T-READ)}$$

$$\frac{P\colon U \quad Q\colon \textsf{end}}{P|Q\colon U} \qquad \frac{P\colon \textsf{end} \quad Q\colon U}{P|Q\colon U} \qquad \text{(T-PAR-L, T-PAR-R)}$$

$$\frac{P\colon U \quad Q\colon \textsf{end}}{\textsf{stream}\, P\, \textsf{as}\, f = \vec{v}\, \textsf{in}\, Q\colon U} \qquad \frac{P\colon \textsf{end} \quad Q\colon U}{\textsf{stream}\, P\, \textsf{as}\, f = \vec{v}\, \textsf{in}\, Q\colon U} \qquad \text{(T-STREAM-L, T-STREAM-R)}$$

$$\frac{P\colon \textsf{end}}{\textsf{rec}\, X.P\colon \textsf{end}} \quad \frac{P\colon U}{(\nu\, n)P\colon U} \quad X\colon \textsf{end} \quad \textsf{0}\colon \textsf{end} \qquad \text{(T-REC, T-RES, T-VAR, T-NIL)}$$

Figure 19: Type system for session sequentiality

5.3 Breaking sequential sessions

The equations proved in the previous section hold for arbitrary processes, and allow us to prove the correctness of the first two optimizations. However to prove the correctness of the implementation step they are not enough. In fact, it is not easy to break a session allowing the conversation to continue in a freshly generated new session, since, in general, communication patterns inside sessions can be quite complex, *e.g.*, since sessions may include many ongoing concurrent communications. However, a small class of sequential sessions captures the most interesting within-session behaviors. Since such a class can be identified by a type system, we start by presenting the type system for *session sequentiality* (similar to the type system for single threadness for mobile safe ambients in [34]), and then we present properties of well-typed processes that allow us to prove the correctness of the last transformation.

The new type system is a simplification of the one previously presented, which guarantees also protocol compatibility. Moreover, we consider here just finite types, thus session protocols should be finite. Notice that this constraint does not forbid infinite behaviors, but just infinite sessions. In particular, if a process is typable according to the general type system and all the involved types are non recursive, then the process is typable according to this type system.

We consider typed processes of the form $P\colon U$ where U is the protocol type. We consider as types $?.U$, $!.U$, and **end**, denoting respectively a protocol that performs an input and then continues as prescribed by U, a protocol that performs an output and then continues as prescribed by U, and the terminated protocol. It is clear that in this setting a request is a session with protocol $!.$**end** (and complementary protocol $?.$**end**), while a request-response has protocol $!.?.$**end** (and complementary protocol $?.!.$**end**).

Definition 13 (Type system for session sequentiality)**.** *The type system is inductively defined by the rules in Figure 19.*

Under the typability assumption, SSCC sessions are sequential in a very strong sense: we can statically define a correspondence between inputs and outputs such that each input is always matched by the corresponding output. We show how to break sessions, *i.e.*, how to make the conversation continue on a freshly created new session. The general law is presented under Theorem 3. The two pieces of the broken session have protocols that are simpler than the original one, thus by repeatedly applying the transformation we can reduce any protocol to a composition of request and request-response patterns. We formalize the procedure described so far.

Definition 14. *Let P be a process. An input/output prefix inside P is* at top-level in *P if it is neither inside a service definition/invocation nor inside a session. Given a process P we can assign sequential indices to top-level input/output prefixes in P according to the position of their occurrence in the term, starting from* 1*. Thus the i*-th top-level prefix in *P is the top-level prefix in P that occurs in i-th position.*

For instance, let P be $a.(x).\textsf{stream}\,(y).\textsf{feed}\,y\,\textsf{as}\,f\ \textsf{in}\,f(y).y\Leftarrow a.(z).\textsf{feed}\,z$. Then P annotated with indices on its top-level prefixes is:

$$a:1.(x):2.\textsf{stream}\,(y):3.\textsf{feed}\,y\,\textsf{as}\,f\ \textsf{in}\,f(y).y\Leftarrow a.(z).\textsf{feed}\,z$$

Definition 15. *Given a process P with a subterm Q, we say that Q is enabled in P if $P = \mathcal{C}[\![Q]\!]$ for some active context $\mathcal{C}[\![\bullet]\!]$. The same definition holds also for prefixes.*

Intuitively a subterm is enabled when it can execute.

Lemma 3. *Let P be a process typable with the type system for session sequentiality. If P has type* **end**, *then it has no top-level enabled prefixes, otherwise it has exactly one top-level enabled prefix, and this has index* 1.

Proof. By induction on the typing proof of P. The thesis follows trivially for rules T-SEND, T-RECEIVE, T-DEF, T-CALL, T-SESS-S, T-SESS-C, T-FEED, T-READ, and T-NIL. For rule T-RES, it follows by inductive hypothesis. For rules T-PAR-L, T-PAR-R, T-STREAM-L, and T-STREAM-R, it follows from the observation that one of the two sides has no enabled prefix, thus the inductive hypothesis can be applied on the other side. □

Definition 16. *Given a transition $P \xrightarrow{\alpha} Q$ and a prefix in P we say that the prefix is consumed if, in the derivation of the transition, rule* L-SEND *or* L-RECEIVE *is applied to the prefix.*

Notice that the consumed prefix does not occur in Q.

Lemma 4. *Let $P_0 = (\nu\, a)\mathcal{D}[\![a\Rightarrow P, a\Leftarrow Q]\!]$ be a typed process such that a does not occur in $\mathcal{D}[\![\bullet,\bullet]\!]$. Suppose $P_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{n-1}} P_n$. Then either:*

- *for all $i \in \{1,\ldots,n\}$ we have $P_i = (\nu\, a)\mathcal{D}'_i[\![a\Rightarrow P, a\Leftarrow Q]\!]$ for some $\mathcal{D}'_i[\![\bullet,\bullet]\!]$ or*
- *there exists $j \in \{1,\ldots,n\}$ such that for each $i < j$ we have $P_i = (\nu\, a)\mathcal{D}'_i[\![a\Rightarrow P, a\Leftarrow Q]\!]$ for some $\mathcal{D}'_i[\![\bullet,\bullet]\!]$, and for each $k \geq j$ we have $P_k = (\nu\, r)\mathcal{D}'_k[\![r\rhd P'_k, r\lhd Q'_k]\!]$ for some $\mathcal{D}'_k[\![\bullet,\bullet]\!]$, processes P'_k and Q'_k and session name r. Furthermore, in a transition $P_k \xrightarrow{\alpha_k} P_{k+1}$ the m-th prefix (according to Definition 14) in P'_k is consumed if and only if the m-th prefix in Q'_k is consumed, and the two are synchronized.*

Proof. The proof is by induction on n. The base case ($n = 0$) is trivial. Let us consider the inductive case. When $P_i = (\nu\, a)\mathcal{D}'_i[\![a\Rightarrow P, a\Leftarrow Q]\!]$ the only possible transitions (since a does not occur in $\mathcal{D}'_i[\![\bullet,\bullet]\!]$) are transitions involving only the context, which leave the process in the same form, or the interaction between the service invocation on a and the service definition of a in the holes. If all the transitions are of the first category then we are in the first case of the lemma. Otherwise, let $j-1$ be the first transition of the second category. This leads to a process of the form $P_j = (\nu\, r)\mathcal{D}'_j[\![r\rhd P'_j, r\lhd Q'_j]\!]$. Let us consider now processes of this form. In order to prove the condition on prefixes we show that the following property holds: at each step either (i) all the prefixes preserve their indices, or (ii) prefixes with index 1 are consumed and the indices of all other prefixes decrease by 1. For transitions not involving prefixes, the thesis follows trivially. Suppose now that, *e.g.*, in P'_k an output prefix is consumed (the case where the prefix is an input prefix or the consumed prefix is in Q'_k is symmetric). Thus $P'_k \xrightarrow{\Uparrow v} P'_{k+1}$ and $r\rhd P'_k \xrightarrow{r\rhd\Uparrow v} r\rhd P'_{k+1}$. Since r is private, this label should interact with a label of the form $r\lhd\downarrow v$. Since sessions are only runtime syntax (cfr. Lemma 5 in Appendix A) such a label can be generated only by $r\lhd Q'_k$, and thanks to Lemma 3 this must be the prefix with index 1. Thus, in both P'_{k+1} and Q'_{k+1} prefix indices are equal to the indices in P'_k and Q'_k minus 1. This proves the thesis. □

We now have all the tools required to prove the correctness of the session breaking technique.

Theorem 3. *Let $(\nu\, a)\mathcal{D}[\![a\Leftarrow \mathcal{C}[\![(x).P]\!], a\Rightarrow \mathcal{C}'[\![v.Q]\!]]\!]$ be a typed process such that a does not occur in $\mathcal{D}[\![\bullet,\bullet]\!]$. Suppose that there exists i such that $(x).P$ and $v.Q$ are the i-th top-level prefixes in $\mathcal{C}[\![(x).P]\!]$ and in $\mathcal{C}'[\![v.Q]\!]$, respectively. Let $y \notin \mathrm{fn}(P)$, $b \notin \mathrm{fn}(Q)$. Then:*

$$(\nu\, a)\mathcal{D}[\![a\Leftarrow \mathcal{C}[\![(x).P]\!], a\Rightarrow \mathcal{C}'[\![v.Q]\!]]\!] \approx_{\mathrm{f}} (\nu\, a)\mathcal{D}[\![a\Leftarrow \mathcal{C}[\![(x,y).y\Leftarrow P]\!], a\Rightarrow \mathcal{C}'[\![(\nu\, b)\langle v,b\rangle.b\Rightarrow Q]\!]]\!]$$

Proof. We show that the following relation is a full weak bisimulation,

$$\begin{gathered}\{((\nu\, a)\mathcal{D}[\![a\Leftarrow \mathcal{C}[\![(x).P]\!], a\Rightarrow \mathcal{C}'[\![v.Q]\!]]\!], \quad (\nu\, a)\mathcal{D}[\![a\Leftarrow \mathcal{C}[\![(x,y).y\Leftarrow P]\!], a\Rightarrow \mathcal{C}'[\![(\nu\, b)\langle v,b\rangle.b\Rightarrow Q]\!]]\!])\\ ((\nu\, r)\mathcal{D}[\![r\lhd \mathcal{C}[\![(x).P]\!], r\rhd \mathcal{C}'[\![v.Q]\!]]\!], \quad (\nu\, r)\mathcal{D}[\![r\lhd \mathcal{C}[\![(x,y).y\Leftarrow P]\!], r\rhd \mathcal{C}'[\![(\nu\, b)\langle v,b\rangle.b\Rightarrow Q]\!]]\!])\\ ((\nu\, r)\mathcal{D}[\![r\lhd \mathcal{C}[\![P]\!], r\rhd \mathcal{C}'[\![Q]\!]]\!], (\nu\, r,b)\mathcal{D}[\![r\lhd \mathcal{C}[\![b\Leftarrow P]\!], r\rhd \mathcal{C}'[\![b\Rightarrow Q]\!]]\!])\\ ((\nu\, r)\mathcal{D}[\![r\lhd \mathcal{C}[\![P]\!], r\rhd \mathcal{C}'[\![Q]\!]]\!], (\nu\, r,r')\mathcal{D}[\![r\lhd \mathcal{C}[\![r'\lhd P]\!], r\rhd \mathcal{C}'[\![r'\rhd Q]\!]]\!])\}\end{gathered}$$

where all the names and processes are universally quantified, $y \notin \text{fn}(P)$, $b \notin \text{fn}(Q)$, and $(x).$ and $v.$ have the same index.

Processes in the first pair can move only to processes of the same shape or to processes in the second pair because of Lemma 4. Similarly, processes in the second pair can move to processes of the same form or to processes in the third pair since prefixes $(x).$ and $v.$ have the same index, and thus are consumed together, again thanks to Lemma 4. In the third pair the only transition that can change the structure of the processes is the invocation of service b on the right, but since this is a τ step, the left part can answer by staying idle. Processes in the last pair only evolve to processes of the same shape. This concludes the proof. □

We are now in a position to prove the soundness of the last transformation.

Soundness of Equation 11. We have to prove that $(\nu\, \mathsf{c})(\mathsf{F} \mid \mathsf{C}) \approx_{\text{f}} (\nu\, \mathsf{c})(\mathsf{G} \mid \mathsf{D})$. For easier reading we recall F, C, G and D below.

$$\mathsf{F} \triangleq b\Rightarrow (x)(\textsf{\textbf{stream}}\; c\Leftarrow x.(z)v.(y)\textsf{\textbf{feed}}\; y.R\; \textsf{\textbf{as}}\; f\; \textsf{\textbf{in}}\; f(y).y.Q)$$
$$\mathsf{C} \triangleq c\Rightarrow (x)w'.(y)v'.S$$
$$\mathsf{G} \triangleq b\Rightarrow (x)(\textsf{\textbf{stream}}\; c\Leftarrow x.(z,c')c'\Leftarrow v.(y)\textsf{\textbf{feed}}\; y.R\; \textsf{\textbf{as}}\; f\; \textsf{\textbf{in}}\; f(y).y.Q)$$
$$\mathsf{D} \triangleq c\Rightarrow (x)(\nu\, c')\langle w',c'\rangle.c'\Rightarrow (y)v'.S$$

It is easy to verify that $(\nu\, \mathsf{c})(\mathsf{F} \mid \mathsf{C})$ can be typed according to the type system for session sequentiality. By considering:

$$\mathcal{D}[\![\bullet_1,\bullet_2]\!] = b\Rightarrow (x)(\textsf{\textbf{stream}}\; \bullet_1\; \textsf{\textbf{as}}\; f\; \textsf{\textbf{in}}\; f(y).y.Q)|\bullet_2$$
$$\mathcal{C}[\![\bullet]\!] = x.\bullet \qquad P = v.(y)\textsf{\textbf{feed}}\; y.R$$
$$\mathcal{C}'[\![\bullet]\!] = (x)\bullet \qquad Q = (y)v'.S$$

we can apply Theorem 3 to get the thesis, since prefixes (z) (in process F) and w' (in process C) have both index 2. □

Notice that the technique above can be extended to break protocols of services with more than one definition/invocation: simply break all of them at the same point.

Let us consider the following example. Let

$$P \triangleq a\Leftarrow v.(x)\textsf{\textbf{stream}}\; b\Leftarrow x.(y).\textsf{\textbf{feed}}\; y\; \textsf{\textbf{as}}\; f\; \textsf{\textbf{in}}\; f(z).z.(w)$$

together with servers for a and b

$$A \triangleq a\Rightarrow (x_1).v_1.(x_2).v_2 \qquad\qquad B \triangleq b\Rightarrow (x_3).v_3$$

It is easy to check that $P\colon$!.?.!.?.**end**, $A\colon$?.!.?.!.**end**, and $B\colon$?.!.**end**.

While B can be easily programmed as a solicit-response (the complementary operation to request-response), P and A require a transformation. Notice that input $(x).$ in P and output $v_1.$ in A have both index 2. We can apply the transformation, thus obtaining processes Q and C equivalent to P and A, respectively, while B is unchanged.

$$Q \triangleq a\Leftarrow v.(x,s)s\Leftarrow \textsf{\textbf{stream}}\; b\Leftarrow x.(y).\textsf{\textbf{feed}}\; y\; \textsf{\textbf{as}}\; f\; \textsf{\textbf{in}}\; f(z).z.(w)$$
$$C \triangleq a\Rightarrow (x_1).(\nu\, c)\langle v_1,c\rangle.c\Rightarrow (x_2).v_2$$

All the services in the new system have type !.?.**end** or ?.!.**end**, thus they can be implemented as request-responses or solicit-responses. The correctness of the transformation, when done on a closed system, is proved by applying Theorem 3.

6 Concluding remarks

SSCC is a typed language aiming at flexibly describing *services*, *conversations*, and *orchestration*, with a restricted set of constructors. The expressiveness of the language is witnessed by the simple implementation of all workflow patterns in [43] (except for the ones that require process termination) and by the examples in Sections 2.1 and 3.2. We have shown instead in Section 5 how to exploit formal techniques to define correct program transformations relating different styles of programming used in the field of service-oriented systems, namely object-oriented, session-based and request/request-response based. This allows to exploit the different techniques available in each field, and still get a system implemented using the desired technology. In addition to that, we have illustrated the expressiveness of SSCC also in a field like object-oriented programming, for which it was not conceived. We have further demonstrated the benefit of working with sequential sessions, where more powerful transformations are available.

As future directions of work, we plan to further study the behavioral theory of SSCC, and to consider other analysis techniques. For the former, we want to investigate the relationships between contextual equivalence and bisimilarity, to look for up-to techniques for bisimilarity, and to try to extend the axioms in Section 4.3 to a complete axiomatization. Concerning the latter, we are interested in more refined techniques for proving service availability (e.g., linearity of service invocation and definition) and in proofs of deadlock freedom for large classes of protocols.

Also, we intend to investigate the possibility of extending the session breaking transformation to larger classes of systems. We are aware of the fact that parallel communications make the agreement between the client and the server on where to change session more difficult. A promising approach to avoid this problem is to perform a preliminary transformation turning arbitrary sessions into sequential ones.

Another thread for future development concerns the definition of a *compensation* mechanism to recover from failures, and the study of its behavioral theory.

Acknowledgements

This work was partially supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the CITI and the LASIGE strategic projects PEst–OE/EEI/UI0527/2011 and PEst–OE/EEI/UI0408/2011 and by the projects PTDC/EIA-CCO/117513/2010 and PTDC/EIA-CCO/122547/2010.

References

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services—Concepts, Architectures and Applications*. Springer, 2003.

[2] Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. *Business Process Execution Language for Web Services*. Version 2.0, 2007.

[3] Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2004.

[4] Michele Banci, Alessandro Fantechi, Michele Ficarra, Silvio Giannini, and Federico Santanni. Automotive case study: a UML description of scenarios. Internal report from the Sensoria EU IST project, 2006.

[5] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[6] Tom Bellwood, Steve Capell, Luc Clément, John Colgrave, Matthew J. Dovey, Daniel Feygin, Andrew Hately, Rob Kochman, Paul Macias, Mirek Novotny, Massimo Paolucci, Claus von Riegen, Tony Rogers, Katia Sycara, Pete Wenzel, and Zhe Wu. *UDDI Version 3.0*, 2004.

[7] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In Paul Gastin and François Laroussinie, editors, *Proc. of CONCUR 2010*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.

[8] Michele Boreale, Roberto Bruni, Luis Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, Antonio Ravara, Davide Sangiorgi, Vasco Vasconcelos, and Gianluigi Zavattaro. SCC: a service centered calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Proc. of WS-FM 2006*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[9] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In Gilles Barthe and Frank S. de Boer, editors, *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[10] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Caspis: A calculus of sessions, pipelines and services. *Mathematical Structures in Computer Science*, 2014. To appear.

[11] Roberto Bruni, Ivan Lanese, Hernán C. Melgratti, and Emilio Tuosto. Multiparty sessions in SOC. In Doug Lea and Gianluigi Zavattaro, editors, *Proc. of COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.

[12] Roberto Bruni and Leonardo G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In José Meseguer and Grigore Rosu, editors, *Proc. of AMAST'08*, volume 5140 of *LNCS*, pages 100–115. Springer, 2008.

[13] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. SOCK: a calculus for service oriented computing. In Asit Dan and Winfried Lamersdorf, editors, *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[14] Luís Caires, Rocco De Nicola, Rosario Pugliese, Vasco T. Vasconcelos, and Gianluigi Zavattaro. Core calculi for service-oriented computing. In Martin Wirsing and Matthias M. Hölzl, editors, *Results of the SENSORIA Project*, volume 6582 of *LNCS*, pages 153–188. Springer, 2011.

[15] Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*, 411(51–52):4399–4440, 2010.

[16] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[17] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Proc. of POPL 2013*, pages 263–274. ACM Press, 2013.

[18] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *WSDL: Web Services Definition Language*. World Wide Web Consortium, 2001.

[19] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In Paolo Ciancarini and Herbert Wiklicky, editors, *Proc. of COORDINATION'06*, volume 4038 of *LNCS*, pages 82–96. Springer, 2006.

[20] Luís Cruz-Filipe, Ivan Lanese, Francisco Martins, António Ravara, and Vasco T. Vasconcelos. Bisimulations in SSCC. DI/FCUL TR 07–37, Department of Informatics, Faculty of Sciences, University of Lisbon, December 2007.

[21] Luís Cruz-Filipe, Ivan Lanese, Francisco Martins, António Ravara, and Vasco T. Vasconcelos. Behavioural theory at work: Program transformations in a service-centred calculus. In Gilles Barthe and Frank de Boer, editors, *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 59–77. Springer, 2008.

[22] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[23] Stefania Gnesi, Maurice ter Beek, Hubert Baumeister, Matthias Hoelzl, Corrado Moiso, Nora Koch, Angelika Zobel, and Michel Alessandrini. D8.0: Case studies scenario description. Deliverable from the Sensoria EU IST project, 2006.

[24] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. *Simple Object Access Protocol (SOAP) 1.2*. World Wide Web Consortium, 2007.

[25] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In Chris Hankin, editor, *Proc. of ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[26] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proc. of POPL'08*, pages 273–284. ACM Press, 2008.

[27] Jolie website. `http://www.jolie-lang.org/`.

[28] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *Proc. of CONCUR'06*, volume 4137 of *LNCS*, pages 477–491. Springer, 2006.

[29] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In Antonio Cerone and Stefan Gruner, editors, *Proc. of SEFM'08*, pages 323–332. IEEE Computer Society, 2008.

[30] Ivan Lanese, António Ravara, and Hugo Torres Vieira. Behavioral theory for session-oriented calculi. In Martin Wirsing and Matthias M. Hölzl, editors, *Results of the SENSORIA Project*, volume 6582 of *LNCS*, pages 189–213. Springer, 2011.

[31] Ivan Lanese, Vasco T. Vasconcelos, Francisco Martins, and António Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM 2007*, pages 305–314. IEEE Computer Society Press, 2007.

[32] Ivan Lanese, Vasco T. Vasconcelos, Francisco Martins, and António Ravara. Disciplining orchestration and conversation in service-oriented computing. DI/FCUL TR 07–3, Department of Informatics, Faculty of Sciences, University of Lisbon, March 2007.

[33] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In Rocco De Nicola, editor, *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[34] Francesca Levi and Davide Sangiorgi. Mobile safe ambients. *ACM Trans. Program. Lang. Syst.*, 25(1):1–69, 2003.

[35] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 6(1):83–110, 2007.

[36] Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad, editors, *Proc. of ICSOC 2011*, volume 7084 of *LNCS*, pages 125–141. Springer, 2011.

[37] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with JOLIE. In *Proc. of ECOWS'07*, pages 13–22. IEEE Computer Society Press, 2007.

[38] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[39] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proc. of TPPP'94*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.

[40] Sensoria project. Software engineering for service-oriented overlay computers. http://www.sensoria-ist.eu/.

[41] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge, 2001.

[42] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *Proc. of PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[43] Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[44] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

[45] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *Proc. of ESOP'08*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

[46] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *Proc. of SecReT'06*, volume 171(4) of *ENTCS*, pages 73–93, 2006.

A Subject reduction

Lemma 5. *For each session name r and each process P, in P there are at most two sessions with name r. If there are exactly two such sessions, then they are one client session $r \lhd P'$ and one server session $r \rhd P''$, they are both in the scope of a binder for r and they are not nested. The three occurrences of r (client, server, and binder) are the only occurrences of r in P.*

Proof. By induction on the length of the computation creating P. The thesis is true for computations of length 0 (sessions do not appear in the static syntax). Sessions are created only by service definitions and service invocations. When a session is created its name is bound, thus different from other session names. Because of that, different service invocations/definitions cannot create sessions with the same name. We have the creation of two sessions with the same name r only when a service invocation on r interacts with the corresponding service definition. In this case the two created sessions are a client session and a server session, they are not nested, and they are both in the scope of a restriction on r. Since r is bound, no other occurrences of r are created in the rest of the computation. □

Lemma 6 (Substitution lemma)**.** *If $\Gamma, x\colon T' \vdash P\colon (U,T)$ and $\Gamma \vdash v\colon T'$ then $\Gamma \vdash P[{}^{v}\!/\!_{x}]\colon (U,T)$.*

Proof. By induction on the typing proof. All the cases are simple. □

Lemma 7. *If $\Gamma \vdash P\colon (\textbf{end},T)$ then P has no transitions of the form $P \xrightarrow{\mu} P'$ with $\mu \in \{\uparrow v, \downarrow v, (v)\uparrow v\}$.*

Proof. The only way to obtain such transitions is to have processes of the form $\mathcal{C}[\![v.P]\!]$ or $\mathcal{C}[\![(x)P]\!]$ where $\mathcal{C}[\![-]\!]$ is composed only by streams, parallel compositions, and restrictions. Let us consider four cases according to the top-level operator in $\mathcal{C}[\![]\!]$.

In the base case we have to use rule T-SEND or T-RECEIVE. These rules do not allow $(\textbf{end},T)$ as resulting type.

In the case of stream we have to use rule T-STREAM-R or T-STREAM-L. We consider just the first case, the second being symmetric. The stream has type $(\textbf{end},T)$ only if the second argument has the same type. Since also the first argument has type $(\textbf{end},T')$ we know by induction that neither of the arguments can do the communication transitions, thus P cannot do them too.

In the case of parallel composition we have to use rule T-PAR-R or T-PAR-L. We consider the first case, the second one being symmetric. Process P has type $(\textbf{end},T)$ only if the second argument has the same type. Since also the first argument has type $(\textbf{end},T)$ we know by induction that neither of the arguments can do the communication transitions, thus P cannot do them too.

In the case of restriction we have to use rule T-RES. Process P has type $(\textbf{end},T)$ only if the restricted process has the same type. We know by induction that the argument cannot do the communication transitions, thus P cannot do them too. □

Lemma 8 (Weakening)**.** *If $\Gamma \vdash P\colon (U,T)$ and $n \notin \mathrm{fn}(P)$ then $\Gamma, n\colon T' \vdash P\colon (U,T)$, for all T'.*

Proof. Simple, by induction on the derivation of the typing judgment. □

Lemma 9 (Strengthening)**.** *If $\Gamma, n : T' \vdash P\colon (U,T)$ and $n \notin \mathrm{fn}(P)$ then $\Gamma \vdash P\colon (U,T)$.*

Proof. Simple, by induction on the derivation of the typing judgment. □

Lemma 10 (Subject congruence)**.** *If $\Gamma \vdash P\colon (U,T)$ and $P \equiv Q$ then $\Gamma \vdash Q\colon (U,T)$.*

Proof. It is enough to show that structural congruent terms can be given the same type using the same assumptions. It is enough to show this for the LHS and the RHS for each structural congruence rule, then the thesis follows by induction (the congruence axioms are simple). All the cases but the one for recursion are easy. We show just this case. Suppose that $\Gamma \vdash \textbf{rec}\, X.P\colon (U,T)$. Then, by hypothesis $\Gamma, X\colon (U,T) \vdash P\colon (U,T)$. By structural induction on P we can prove that if $\Gamma, X\colon (U,T) \vdash P\colon (U,T)$ then $\Gamma \vdash P[{}^{\textbf{rec}\, X.P}\!/\!_{X}]\colon (U,T)$. This holds for the case of $P = X$ and is preserved by all the contexts (notice in fact that the assumptions about different occurrences of the same variable are compatible). The proof is similar in the opposite direction. □

Let $\Gamma[{}^{[U']}\!/\!_{r}]$ denote the substitution on Γ of $[U']$ for $\Gamma(r)$.

Theorem 4. *Let P be a process such that $\Gamma \vdash P\colon (U,T)$. Then:*

- *if $P \xrightarrow{\upharpoonright v} P'$ then $U = !T'.U'$, $\Gamma \vdash v\colon T'$ and $\Gamma \vdash P'\colon (U',T)$;*
- *if $P \xrightarrow{(v)\upharpoonright v} P'$ then $U = !T'.U'$ and $\Gamma, v\colon T' \vdash P'\colon (U',T)$;*
- *if $P \xrightarrow{\downharpoonright v} P'$ then $U = ?T'.U'$ and $\Gamma, v\colon T' \vdash P'\colon (U',T)$;*
- *if $P \xrightarrow{a \Leftarrow (r)} P'$ then $\Gamma \vdash a\colon [U']$ and $\Gamma, r\colon [U'] \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{a \Rightarrow (r)} P'$ then $\Gamma \vdash a\colon [U']$ and $\Gamma, r\colon [U'] \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{\Uparrow v} P'$ then $\Gamma \vdash v\colon T$ and $\Gamma \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{(v)\Uparrow v} P'$ then $\Gamma, v\colon T \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{f \Downarrow v} P'$ then $\Gamma \vdash f\colon \langle T \rangle$ and $\Gamma, v\colon T \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{r \rhd \upharpoonright v} P'$ then $\Gamma \vdash r\colon [!T'.U']$, $\Gamma \vdash v\colon T'$ and $\Gamma[{}^{[U']}\!/\!_{r}] \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{(v) r \rhd \upharpoonright v} P'$ then $\Gamma \vdash r\colon [!T'.U']$ and $\Gamma[{}^{[U']}\!/\!_{r}], v\colon T' \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{r \rhd \downharpoonright v} P'$ then $\Gamma \vdash r\colon [?T'.U']$ and $\Gamma[{}^{[U']}\!/\!_{r}], v\colon T' \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{r \lhd \upharpoonright v} P'$ then $\Gamma \vdash r\colon [?T'.U']$, $\Gamma \vdash v\colon T'$ and $\Gamma[{}^{[U']}\!/\!_{r}] \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{(v) r \lhd \upharpoonright v} P'$ then $\Gamma \vdash r\colon [?T'.U']$ and $\Gamma[{}^{[U']}\!/\!_{r}], v\colon T' \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{r \lhd \downharpoonright v} P'$ then $\Gamma \vdash r\colon [!T'.U']$ and $\Gamma[{}^{[U']}\!/\!_{r}], v\colon T' \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{r\tau} P'$ then $\Gamma \vdash r\colon [!T'.U']$ or $\Gamma \vdash r\colon [?T'.U']$ and $\Gamma[{}^{[U']}\!/\!_{r}] \vdash P'\colon (U,T)$;*
- *if $P \xrightarrow{\tau} P'$ then $\Gamma \vdash P'\colon (U,T)$.*

Proof. The proof is by induction on the derivation of the transition. We perform a case analysis on the last rule used in the derivation.

L-SEND: P has the form $v.P'$. This can be typed only using rule T-SEND and this requires $U = !T'.U'$, $\Gamma \vdash P'\colon (U',T)$ and $\Gamma \vdash v\colon T'$. This is exactly as desired.

L-RECEIVE: P has the form $(x)P''$ and $P' = P''[{}^{v}\!/\!_{x}]$. P can be typed only using rule T-RECEIVE and this requires $U = ?T'.U'$ and $\Gamma, x\colon T' \vdash P'\colon (U',T)$. Thanks to Lemma 6 we also have $\Gamma, v\colon T' \vdash P'[{}^{v}\!/\!_{x}]\colon (U',T)$.

L-CALL: P has the form $a \Leftarrow P''$ and $P' = r \lhd P''$. P can be typed only using rule T-CALL and this requires $U = \textsf{end}$, $\Gamma \vdash P'' : (U',T)$ and $\Gamma \vdash a : [\overline{U'}]$. Using rule T-SESS-C (and thanks to Lemma 8) one can derive $\Gamma, r\colon [\overline{U'}] \vdash r \lhd P''\colon (\textsf{end}, T)$.

L-DEF: P has the form $a \Rightarrow P''$ and $P' = r \rhd P''$. P can be typed only using rule T-DEF and this requires $U = \textsf{end}$, $\Gamma \vdash P'' : (U',T)$ and $\Gamma \vdash a : [U']$. Using rule T-SESS-S (and thanks to Lemma 8) one can derive $\Gamma, r\colon [U'] \vdash r \rhd P''\colon (\textsf{end}, T)$.

L-FEED: P has the form $\textsf{feed}\, v.P'$. This can be typed only using rule T-FEED and this requires $\Gamma \vdash P'\colon (U,T)$ and $\Gamma \vdash v\colon T$. This is exactly as required.

L-READ: P has the form $f(x).P''$ and $P' = P''[{}^{v}\!/\!_{x}]$. P can be typed only using rule T-READ and this requires $\Gamma, x\colon T' \vdash P''\colon (U,T)$ and $\Gamma \vdash f\colon \langle T'\rangle$. From Lemma 6 we have $\Gamma, v\colon T' \vdash P''[{}^{v}\!/\!_{x}]\colon (U,T)$ as required.

L-STREAM-PASS-P: P has the form $\textbf{stream}\, P''\, \textbf{as}\, f = \vec{v}\, \textbf{in}\, Q$ with $P'' \xrightarrow{\mu} P'''$ and we have $P' = \textbf{stream}\, P'''\, \textbf{as}\, f = \vec{v}\, \textbf{in}\, Q$. There are two cases according to the last rule used to type P. We consider rule T-STREAM-R first and rule T-STREAM-L later. Thanks to Lemma 7, $\mu \notin \{\uparrow v, \downarrow v, (v) \uparrow v\}$. Also, $\mu \notin \{\Uparrow v, (v) \Uparrow v\}$. By hypothesis all the assumptions on f, $\vec{v}$ and Q are satisfied. By induction hypothesis in all the cases but $r \bowtie\!\uparrow v$, $(v)r \bowtie\!\uparrow v$, $r \bowtie\!\downarrow v$ and $r\tau$ we have that $\Gamma' \vdash P'''\colon (\textbf{end}, T)$ for some extension Γ' of Γ. Thanks to Lemma 8, Γ' can be used to derive $\Gamma' \vdash P'\colon (U,T)$ as required. Notice also that the assumptions on Γ' are satisfied by induction hypothesis since the label is unchanged. For the other cases the problem is that the assumption about r is changed. However, thanks to Lemma 5 there are two cases. If there is just one occurrence of r, thus the assumption is never used outside P''', Lemma 9 can be used to drop the old assumption and Lemma 8 to add the new one, and the thesis follows. If there are three occurrences two should be in opposite session constructs and the third in a restriction binding them. The only label of these that can cross the restriction is $r\tau$, thus no occurrence of r can be in Q, since otherwise we can not obtain this label. Thus r is not used in Q and we can derive $\Gamma' \vdash Q\colon (U,T)$ as required, using again lemmas 9 and 8. Thus we can also derive $\Gamma' \vdash P'\colon (U,T)$ and the thesis follows.

Let us consider the second case. Notice that $\mu \notin \{\Uparrow v, (v) \Uparrow v\}$. Now both U and μ are preserved from the premise, thus in most of the cases the thesis follows immediately from the induction premise (when a new assumption is needed in Γ, such as in extrusions, Lemma 8 can be used, and the compatibility of the new assumption is guaranteed by the side condition on bound names of the typing rule). The only tricky cases concern labels $r \bowtie\!\uparrow v$, $(v)r \bowtie\!\uparrow v$, $r \bowtie\!\downarrow v$ and $r\tau$, but the same reasoning above applies. The thesis follows.

L-STREAM-PASS-Q: P has the form $\textbf{stream}\, P''\, \textbf{as}\, f = \vec{v}\, \textbf{in}\, Q$ with $Q \xrightarrow{\mu} Q'$ and we have $P' = \textbf{stream}\, P''\, \textbf{as}\, f = \vec{v}\, \textbf{in}\, Q'$. By hypothesis all the assumptions on P, f and $\vec{v}$ are satisfied. Also, $\Gamma, f : \langle T'\rangle \vdash Q\colon (U,T)$. By induction hypothesis $\Gamma', f : \langle T'\rangle \vdash Q'\colon (U',T)$ where Γ' and U' are defined by the statement of the theorem. Notice that Γ' verifies all the assumptions of rule T-STREAM-L (resp. T-STREAM-R) since it is either an extension of Γ (and in this case Lemma 8 can be used), or it changes the assumption about some session r, and in this case the same reasoning done for rule L-STREAM-PASS-P can be used. Thus one can use rule T-STREAM-L (resp. T-STREAM-R) to derive $\Gamma' \vdash P'\colon (U',T)$ as required.

L-STREAM-FEED: P has the form $\textbf{stream}\, P''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q$ with $P'' \xrightarrow{\Uparrow v} P'''$ and we have $P' = \textbf{stream}\, P'''\, \textbf{as}\, f = v :: \vec{w}\, \textbf{in}\, Q$. There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider the first one, the second being similar. By hypothesis $\Gamma \vdash P''\colon (\textbf{end}, T')$, $\Gamma, f\colon \langle T'\rangle \vdash Q\colon (U,T)$ and $w' \in \mathrm{Set}(\vec{w}) \Rightarrow \Gamma \vdash w'\colon T'$. By induction hypothesis $\Gamma \vdash v\colon T'$ and $\Gamma \vdash P'''\colon (\textbf{end}, T')$. Thus using rule T-STREAM-R we can prove $\Gamma \vdash P'\colon (U,T)$ (notice, in particular, that the assumption about $v :: \vec{w}$ can be proved from the assumptions about v and $\vec{w}$).

L-STREAM-CONS: P has the form $\textbf{stream}\, P''\, \textbf{as}\, f = \vec{w} :: v\, \textbf{in}\, Q$ with $Q \xrightarrow{f \Downarrow v} Q'$ and we have $P' = \textbf{stream}\, P''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q'$. There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider the first one, the second being symmetric. By hypothesis $\Gamma \vdash P''\colon (\textbf{end}, T')$, $\Gamma, f\colon \langle T'\rangle \vdash Q\colon (U,T)$ and $w' \in \mathrm{Set}(\vec{w} :: v) \Rightarrow \Gamma \vdash w'\colon T'$. By induction hypothesis $\Gamma, f\colon \langle T'\rangle, v\colon T' \vdash Q'\colon (U,T)$. Since $\Gamma, v\colon T'$ is an extension of Γ we can use it (thanks to Lemma 8) in all the premises of rule T-STREAM-R and finally derive $\Gamma, v\colon T' \vdash P'\colon (U,T)$.

L-PAR: the reasoning is as for rule L-STREAM-PASS-P, but there is no stream here.

L-SESS-VAL: we consider just the cases for $\lhd$, the other being simpler. P has the form $r \lhd P''$. By hypothesis $U = \textbf{end}$, $\Gamma \vdash P''\colon (U',T)$ and $\Gamma \vdash r\colon [\overline{U'}]$. Let us consider the case $P'' \xrightarrow{\uparrow v} P'''$ before. This implies $P' = r \lhd P'''$. By induction hypothesis $U' = !T'.U''$, $\Gamma \vdash v\colon T'$ and $\Gamma \vdash$

$P''' \colon (U'', T)$. Using rule T-SESS-C we can prove $\Gamma[{}^{[\overline{U''}]}\!/_{r}] \vdash r \lhd P''' \colon (\textbf{end}, T)$ as required since this is the only place where the assumption about r is used inside the term thanks to Lemma 5, thus it can be changed using lemmas 9 and 8. Let us now consider the case $P'' \xrightarrow{\Downarrow v} P'''$. Again $P' = r \lhd P'''$. By induction hypothesis $U' = ?T'.U'', \Gamma, v \colon T' \vdash P''' \colon (U'', T)$. Using rule T-SESS-C we can prove $\Gamma[{}^{[\overline{U''}]}\!/_{r}], v \colon T' \vdash r \lhd P'''$ as required, since this is the only place where the assumption about r is used inside the term thanks to Lemma 5, thus it can be changed using lemmas 9 and 8.

L-SESS-PASS: we consider just the cases for $\lhd$, the others being simpler. P has the form $r \lhd P''$ with $P'' \xrightarrow{\mu} P'''$ and $P' = r \lhd P'''$. By hypothesis $U = \textbf{end}, \Gamma \vdash P'' \colon (U', T)$ and $\Gamma \vdash r \colon [\overline{U'}]$. Notice that $\mu \neq\!\updownarrow v$. Thus for all the cases but session communication labels we have $\Gamma' \vdash P''' \colon (U', T)$ for some extension Γ' of Γ. In the case of session communication labels instead the assumption about r' is changed from Γ to Γ'. Notice that thanks to Lemma 5 $r \neq r'$, thus in both the cases we can use rule T-SESS-C to derive $\Gamma' \vdash r \lhd P''' \colon (\textbf{end}, T)$ as required, since the label of the new transition is equal to the label of the premise, thus the assumptions on Γ' coincide.

L-SESS-COM-STREAM: P has the form $\textbf{stream}\, P''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q$ with $P'' \xrightarrow{r \rhd \Uparrow v} P''', Q \xrightarrow{r \lhd \Downarrow v} Q'$ and $P' = \textbf{stream}\, P'''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q'$ (the other cases are similar). There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider just the first one, the second being similar. By hypothesis $\Gamma \vdash P \colon (\textbf{end}, T')$ and $\Gamma, f \colon \langle T' \rangle \vdash Q \colon (U, T)$. By induction hypothesis on the first transition $\Gamma \vdash r \colon [!T''.U'], \Gamma \vdash v \colon T''$ and $\Gamma[{}^{[U']}\!/_{r}] \vdash P''' \colon (U, T)$. From the second transition we have a redundant hypothesis on r and $\Gamma[{}^{[U']}\!/_{r}], f \colon \langle T' \rangle, v \colon T' \vdash Q' \colon (U, T)$. Notice that $\Gamma[{}^{[U']}\!/_{r}], f \colon \langle T' \rangle, v \colon T' = \Gamma[{}^{[U']}\!/_{r}], f \colon \langle T' \rangle$ since $\Gamma[{}^{[U']}\!/_{r}], f \colon \langle T' \rangle \vdash v \colon T'$. Thus we can apply rule T-STREAM-R to derive $\Gamma[{}^{[U']}\!/_{r}] \vdash P' \colon (U, T)$ as required.

L-SERV-COM-STREAM: P has the form $\textbf{stream}\, P''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q$ with $P'' \xrightarrow{a \Rightarrow (r)} P''', Q \xrightarrow{a \Leftarrow (r)} Q'$ and $P' = (\nu\, r)\textbf{stream}\, P'''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q'$ (the symmetric case is similar). There are two cases corresponding to rules T-STREAM-R and T-STREAM-L. We consider just the first one, the second being similar. By hypothesis $\Gamma \vdash P'' \colon (\textbf{end}, T')$ and $\Gamma, f \colon \langle T' \rangle \vdash Q \colon (U, T)$. By induction hypothesis (on both the transitions) $\Gamma \vdash a \colon [U']$ and $\Gamma, r \colon [U'] \vdash P'' \colon (\textbf{end}, T')$ and $\Gamma, f \colon \langle T' \rangle, r \colon [U'] \vdash Q' \colon (U, T)$. Using rule T-STREAM-R we can derive the judgment $\Gamma, r \colon [U'] \vdash \textbf{stream}\, P'''\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, Q' \colon (U, T)$. Then we can use rule T-RES to derive $\Gamma \vdash P' \colon (U, T)$ as desired.

L-SESS-COM-PAR: the reasoning is as for rule L-SESS-COM-STREAM, but there is no stream here.

L-SERV-COM-PAR: the reasoning is as for rule L-SERV-COM-STREAM, but there is no stream here.

L-RES: P has the form $(\nu\, n)P''$ with $P'' \xrightarrow{\mu} P'''$ and $P' = (\nu\, n)P'''$. By hypothesis $\Gamma, n \colon T' \vdash P'' \colon (U, T)$ for some T'. By induction hypothesis $\Gamma', n \colon T' \vdash P''' \colon (U', T)$ where Γ' and U' are as defined by the statement of the theorem. Thus we can apply rule T-RES to derive $\Gamma' \vdash (\nu\, n)P''' \colon (U', T)$ since the label is unchanged thus Γ' and U' are as before.

L-EXTR: P has the form $(\nu\, a)P''$ with $P'' \xrightarrow{\mu} P'$. By hypothesis $\Gamma, a \colon T' \vdash P'' \colon (U, T)$. Thanks to the induction hypothesis $\Gamma', a \colon T' \vdash P' \colon (U', T)$ where Γ' and U' are as described in the statement of the theorem. This is exactly as required, given the different requirements between each action and the corresponding extruding action.

L-SESS-RES: P has the form $(\nu\, r)P''$ with $P'' \xrightarrow{r\tau} P'''$ and $P' = (\nu\, r)P'''$. By hypothesis $\Gamma, r \colon [U'] \vdash P'' \colon (U, T)$ (the type of r should be a protocol since r is a session). By the induction hypothesis $\Gamma, r \colon [U''] \vdash P''' \colon (U, T)$. Then we can use rule T-RES to derive $\Gamma \vdash P''' \colon (U, T)$ as required.

L-STRUCT: By Lemma 10.

□

Proof of Theorem 1, page 13 (Subject Reduction). The thesis follows from Theorem 4 and the characterization of reductions as transitions with labels τ given in Theorem 2. □

Proof of Theorem 2, page 13 (Type Safety). The proofs of all the cases are by contradiction. We suppose that such a subterm exists and we show that it is not typable. We consider the two different cases:

Protocol: Let us consider the first case. Here $v.P$ and $u.Q$ have types of the form $([!T.U], T'')$ and $([!T'.U'], T''')$ respectively. One can prove by structural induction on the context that the protocol part of the type is preserved (only the session construct can change it, but the side condition forbids sessions around the hole). Thus the two session constructs require $r\colon [!T.U]$ and $r\colon [?T'.\overline{U'}]$ (supposing that the first one is a server session, the symmetric otherwise). Since $\mathcal{D}[\![,]\!]$ does not bind r the assumptions are preserved, and at top level they should agree since the same Γ is used to type the two sides of parallel composition or stream. This is not the case and we have the required contradiction. The other cases are similar, with just **end** protocol for **0** and $([?T.U], T'')$ for input.

Sequentiality: In all the cases the two terms inserted into the double context have non **end** protocol. The property is preserved by the context (since there are no sessions around the hole). At top level we have two non **end** protocols, but the rules for parallel composition and stream can not be applied because of this. Since no other rules can type a parallel composition or a stream we have the desired contradiction.

$\square$

B Workflow patterns in SSCC

This section completes Section 3.1, by modeling in SSCC the remaining workflow patterns from [43].

WP2: Parallel Split

"A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order. Example: after registering an insurance claim, two parallel subprocesses are triggered: one for checking the policy of the customer and one for assessing the actual damage."

Parallel composition is built-in. The same in SCC and in Orc.

WP5: Simple Merge

"A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel. Example: after the payment is received or the credit is granted, the car is delivered to the customer."

```
merge  ::  Bool → (ε → T) →...→ Bool → (ε → T) → Unit
merge  ⇛  (b1)(a1)...(bn)(an)
           (if b1 then call a1 | ... | if bn then call an) >1> unit
```

This is more in line with van der Aalst specification than the corresponding model in Orc, since SSCC is able to model the fact that only some of the activities are activated. Notice that, given the assumptions, replacing $>^{\mathbf{1}}>$ with $>>$ will not change the behavior.

WP6: Multi-Choice

"A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen. Example: after executing the activity *evaluate_damage*, the activity *contact_fire_department* or the activity *contact_insurance_company* is executed. At least one of these activities is executed. However, it is also possible that both need to be executed."

Notice that the code resulting from the application of this WP is not an activity, since it may produce more than one result. The more natural implementation of this WP in SSCC is:

```
multiChoice ⇛ (b₁)(a₁)...(bₙ)(aₙ)
              (if b₁ then call a₁ >¹ x > x | ... |
               if bₙ then call aₙ >¹ x > x)
```

A similar implementation is possible in Orc and in SCC.

However, this service is not typable, since all the results are sent in parallel inside the same session. To solve this problem we can sequentialize them by writing:

```
multiChoice :: [?Bool.?(ε → T). ... .?Bool.?(ε → T).!T. ... !T.end]
multiChoice ⇛ (b₁)(a₁)...(bₙ)(aₙ)
               stream
              (if b₁ then call a₁ | ... | if bₙ then call aₙ)
               as f in
               f(y₁).y₁ .... f(yₙ).yₙ
```

Note that the service may not produce the number of results specified by its type, since the stream may not supply enough elements.

WP7: Synchronizing Merge

"A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronisation of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete. Example: extending the example of WP6 (Multi-choice), after either or both of the activities *contact_fire_department* and *contact_insurance_company* have been completed (depending on whether they were executed at all), the activity *submit report* needs to be performed (exactly once)."

```
syncMerge:: (ε → Bool) → (ε → Unit) →...→
            (ε → Bool) → (ε → Unit) → Unit
syncMerge ⇛ (b₁)(a₁)...(bₙ)(aₙ)
            call sync(ifSignal_b₁_a₁,...,ifSignal_bₙ_aₙ) >ⁿ > unit
ifSignal_bᵢ_aᵢ :: ε → Unit
ifSignal_bᵢ_aᵢ ⇒ IfSignal(bᵢ, call aᵢ >¹ x > x)
```

where

```
IfSignal(b,P) = if b then P else unit
```

Essentially, we reuse WP3 (Synchronization) on services ifSignal_b_i_a_i. Service ifSignal_b_i_a_i invokes a_i and gives back its result if b_i is true, it immediately returns **unit** otherwise. When all the results from the invoked services have been collected, a **unit** value is returned. It can be used to trigger the final activity.

Similar to Orc and SCC.

WP8: Multi-Merge

"A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started *for every activation of every incoming branch*. Example: two activities *audit_application* and *process_application* running in parallel, which should both be followed by an activity *close_case*."

```
merge :: [?Bool.?(ε → T). ... .?Bool.?(ε → T).?(ε → T1).!T1.....!T1.end]
merge ⇛ (b₁)(a₁)...(bₙ)(aₙ)(c)
        stream
        (if b₁ then call a₁ | ... | if bₙ then call aₙ)
        as f in
           stream
           f(y₁).call c(y₁)....f(yₙ).call c(yₙ)
           as g in
           g(z₁).z₁.....g(zₙ).zₙ
```

This is not an activity, since it provides multiple replies. Similar to the Orc implementation. In SCC one can use the technique of WP1 (Sequence). Notice also that now the behavior of the synchronization is the expected one (one instance is launched for each value).

WP11: Implicit Termination

"A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock)."

This pattern is not meaningful in process calculi. In fact, in calculi the standard behavior is that processes naturally terminate when they have finished their activity, not when a final state is reached by one of their components. This is the case for both SCC and SSCC, while this is not the case in workflow managers.

WP12: Multiple Instances without Synchronization

"Within the context of a single case (*i.e.*, workflow instance) multiple instances of an activity can be created, *i.e.*, there is a facility to spawn new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronise these threads. Example: a customer ordering a book from an electronic bookstore such as Amazon may order multiple books at the same time. Many of the activities (*e.g.*, billing, updating customer records) occur at the level of the order. However, within the order, multiple instances need to be created to handle the activities related to one individual book (*e.g.*, update stock levels, shipment). If the activities at the book level do not need to be synchronized, this pattern can be used."

Multiple instances of the same service can be executed concurrently without any particular problem. An unbounded number of instances can be created by persistent services. The same in SCC.

WP13: Multiple Instances with a Priory Design Time Knowledge

"For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is known at design time. Once all instances are completed, some other activity needs to be started. Example: the requisition of hazardous material requires three different authorizations."

```
sync_n :: (ε → T) → Unit
sync_n ⤃ (a) call sync (a,... ,a) >¹ x > x
```

Since the number of instances (calls to) of service a is known to be n, it is enough to pass n arguments to service sync (cfr. WP3: Synchronization).

A similar approach can be used in Orc. A possible implementation in SCC is:

```
sync_n ⇒ (a) sync {a. ... .a.(x) return x} ⇐ a
```

where one instance of a is passed as invocation parameter, and the orher n−1 instances are passed inside the session protocol.

WP14: Multiple Instances with a Priory Run-time Knowledge

"For one case, an activity is enabled multiple times. The number of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources, but is known at some stage during run-time, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started. Example: when booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client."

We treat this case as a particular case of WP15. See below for the discussion.

WP15: Multiple Instances without a Priory Run-time Knowledge

“For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is not known during design time, nor is it known at any stage during run-time, before the instances of that activity have to be created. Once all instances are completed, some other activity needs to be started. The difference with WP14 is that even while some of the instances are being executed or already completed, new ones can be created. Example: for the processing of an insurance claim, zero or more eyewitness reports should be handled. The number of eyewitness reports may vary. Even when processing eyewitness reports for a given insurance claim, new eyewitnesses may surface and the number of instances may change.”

Invoke service a as long as service c replies true. Instances are executed in parallel: the first instance is launched in parallel with parloop_c_a. Termination of an instance is checked together with the termination of the parloop launched together.

```
parloop_c_a :: ε → Unit
parloop_c_a ⤔⇒ call c >¹ b >
               IfSignal(b, call sync(a,parloop_c_a)) >¹ x > x
```

For simplicity we have chosen a loop service specific for c and a. To write a generic loop service that accepts two parameters (c and a) we have to customize sync to invoke services with parameters. We leave the exercise to the reader. Similar implementations can be done in Orc and in SCC.

As far as WP14 is concerned, the main choice is how to represent the run-time knowledge about the required number of instances to be executed, *i.e.*, how to represent state. Possibilities include taking advantage of the number of values in a stream, of the number of available instances of a service, or of the number of values in a session protocol.

WP16: Deferred Choice

“A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (*e.g.*, based on data or on a decision), but several alternatives are offered to the environment. However, in contrast to the AND-split, only one of the alternatives is executed. This means that once the environment activates one of the branches, the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, *i.e.*, the moment of choice is as late as possible. Example: after receiving products there are two ways to transport them to the department. The selection is based on the availability of the corresponding resources. Therefore, the choice is deferred until a resource is available.”

Requires a means to kill unwanted computations, such as the **where** operator in Orc.

C On the labeled transition system of SSCC

C.1 The Harmony Lemma

In the proof below we use $\overline{a}$ to denote any label in $\{\uparrow a, r \bowtie\uparrow a, \Uparrow a\}$.

Theorem 5 (Harmony Lemma)**.** *Let P and Q be processes with $P \equiv Q$. If $P \xrightarrow{\alpha} P'$, then $Q \xrightarrow{\alpha} Q'$ with $P' \equiv Q'$, and vice-versa.*

Proof. By induction on the proof that $P \equiv Q$.

- *Equivalence relation*
  - Reflexivity. Immediate, taking Q' to be P'.
  - Symmetry. Immediate consequence of the induction hypothesis, since the thesis of the theorem is symmetric.

- Transitivity. Assume $P \equiv Q$ because $P \equiv R$ and $R \equiv Q$, and suppose that $P \xrightarrow{\alpha} P'$. By induction hypothesis, $R \xrightarrow{\alpha} R'$ with $P' \equiv R'$; hence, again by induction hypothesis, $Q \xrightarrow{\alpha} Q'$ with $R' \equiv Q'$. $P' \equiv Q'$ follows by transitivity of $\equiv$.

- *Congruence properties*
  - Parallel composition. Suppose $P \equiv Q$. For each of the possible transitions of $P \mid R$, it is straightforward to verify that $Q \mid R$ can simulate them, possibly using the induction hypothesis; similarly, $R \mid Q$ can simulate $R \mid P$. Notice that the side conditions in the transition rules always hold since structurally congruent processes have the same free names.
  - Composition with stream. Analogous.
  - Name restriction. Suppose $P \equiv Q$ and let a be a name. For each of the possible transitions of $(\nu a)P$, it is easy to check that $(\nu a)Q$ can simulate them, possibly using the induction hypothesis.
  - Session input/output. Straightforward, observing (for input) that structural congruence is closed under substitution.
  - Stream input/output. Analogous.
  - Service definition/invocation. Straightforward.

- *Monoid structure*
  - Unit. Let Q be $P \mid 0$. If $P \xrightarrow{\alpha} P'$, then by rule L-PAR also $P \mid 0 \xrightarrow{\alpha} P' \mid 0$, since 0 has no free names. Also, $P' \mid 0$ is congruent to P'. Reciprocally, if $P \mid 0 \xrightarrow{\alpha} P$', then the only rule that can have been applied is L-PAR (since $0 \not\rightarrow$), whence P' is $P'' \mid 0$ with $P \xrightarrow{\alpha} P''$.
  - Commutativity. Assume P is $R \mid S$ and Q is $S \mid R$. Take any proof of $R \mid S \xrightarrow{\alpha} T$ and replace occurrences of L-PAR by L-PAR', of L-SESS-COM-PAR by L-SESS-COM-PAR', of L-SERV-COM-PAR by L-SERV-COM-PAR' and vice-versa; it is straightforward to verify that this yields a proof that $S \mid R \xrightarrow{\alpha} T'$ with $T \equiv T'$. The converse is analogous.
  - Associativity. Let P be $R \mid (S \mid T)$ and Q be $(R \mid S) \mid T$. Suppose that $P \xrightarrow{\alpha} P'$; there are six rules that can be used to infer this transition. For simplicity, in the proofs below we omit side conditions related to bound names, since it is simple to verify that they always follow from the assumptions.
    - L-PAR: then $R \xrightarrow{\alpha} R'$ and P' is $R' \mid (S \mid T)$. The proof below shows that $(R \mid S) \mid T \xrightarrow{\alpha} (R' \mid S) \mid T$, which establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{\alpha} R'}{R \mid S \xrightarrow{\alpha} R' \mid S}\ \text{L-PAR}}{(R \mid S) \mid T \xrightarrow{\alpha} (R' \mid S) \mid T}\ \text{L-PAR}$$

    - L-PAR': then $S \mid T \xrightarrow{\alpha} U$; there are six sub-cases, according to the rule used to derive this transition.
      - The rule applied is L-PAR, so $S \xrightarrow{\alpha} S'$ and U is $S' \mid T$; then the following proof establishes the thesis.

$$\dfrac{\dfrac{S \xrightarrow{\alpha} S'}{R \mid S \xrightarrow{\alpha} R \mid S'}\ \text{L-PAR'}}{(R \mid S) \mid T \xrightarrow{\alpha} (R \mid S') \mid T}\ \text{L-PAR}$$

      - The rule applied is L-PAR', so $T \xrightarrow{\alpha} T'$ and U is $S \mid T'$; then the following proof establishes the thesis.

$$\dfrac{T \xrightarrow{\alpha} T'}{(R \mid S) \mid T \xrightarrow{\alpha} (R \mid S) \mid T'}\ \text{L-PAR'}$$

· The rule applied is L-SESS-COM-PAR, so $S \xrightarrow{r\bowtie\Updownarrow v} S'$, $T \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} T'$, U is $S' \mid T'$ and α is $r\tau$ for some fresh r; then the following proof establishes the thesis.

$$\dfrac{\dfrac{S \xrightarrow{r\bowtie\Updownarrow v} S'}{R \mid S \xrightarrow{r\bowtie\Updownarrow v} R \mid S'}\,\text{L-PAR'} \qquad T \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} T'}{(R \mid S) \mid T \xrightarrow{r\tau} (R \mid S') \mid T'}\,\text{L-SESS-COM-PAR}$$

· The rule applied is L-SERV-COM-PAR, thus we have $S \xrightarrow{a\Leftrightarrow(r)} S'$, $T \xrightarrow{a\overline{\Leftrightarrow}(r)} T'$, U is $(\nu r)(S' \mid T')$ and α is τ; then the following proof establishes the thesis, since $(\nu r)((R \mid S') \mid T') \equiv R \mid ((\nu r)(S' \mid T'))$ as r is not a free name of R.

$$\dfrac{\dfrac{S \xrightarrow{a\Leftrightarrow(r)} S'}{R \mid S \xrightarrow{a\Leftrightarrow(r)} R \mid S'}\,\text{L-PAR'} \qquad T \xrightarrow{a\overline{\Leftrightarrow}(r)} T'}{(R \mid S) \mid T \xrightarrow{\tau} (\nu r)((R \mid S') \mid T')}\,\text{L-SERV-COM-PAR}$$

· The rule applied is L-PAR-CLOSE, so $S \xrightarrow{r\bowtie(a)\overline{a}} S'$, $T \xrightarrow{r\overline{\bowtie}\downarrow a} T'$, U is $(\nu a)(S' \mid T')$ and α is $r\tau$. This case is analogous to that of L-SESS-COM-PAR, the extra name restrictions in the resulting processes posing no additional problem.

· The rule applied is L-PAR-CLOSE', so $S \xrightarrow{r\bowtie\downarrow a} S'$, $T \xrightarrow{r\overline{\bowtie}(a)\overline{a}} T'$, U is $(\nu a)(S' \mid T')$ and α is $r\tau$. This case is analogous to the previous one.

∗ L-SESS-COM-PAR: then $R \xrightarrow{r\bowtie\Updownarrow v} R'$ and $S \mid T \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} U$; there are two similar sub-cases, according to whether the last transition is proved via L-PAR or via L-PAR'. Without loss of generality, assume that the former is the case; then the following proof establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{r\bowtie\Updownarrow v} R' \quad S \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} S'}{R \mid S \xrightarrow{r\tau} R' \mid S'}\,\text{L-SESS-COM-PAR}}{(R \mid S) \mid T \xrightarrow{r\tau} (R' \mid S') \mid T}\,\text{L-PAR'}$$

∗ L-SERV-COM-PAR: then $R \xrightarrow{a\Leftrightarrow(r)} R'$ and $S \mid T \xrightarrow{a\overline{\Leftrightarrow}(r)} U$; again there are two similar sub-cases, according to whether the last transition is proved via L-PAR or via L-PAR'. Without loss of generality, assume that the former is the case; then the following proof establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{a\Leftrightarrow(r)} R' \quad S \xrightarrow{a\overline{\Leftrightarrow}(r)} S'}{R \mid S \xrightarrow{\tau} (\nu r)(R' \mid S')}\,\text{L-SERV-COM-PAR}}{(R \mid S) \mid T \xrightarrow{\tau} (\nu r)(R' \mid S') \mid T}\,\text{L-PAR'}$$

Since r is not a free name of T, the latter process is structurally congruent to process $(\nu r)(R' \mid (S' \mid T))$.

∗ L-PAR-CLOSE: then $R \xrightarrow{r\bowtie(a)\overline{a}} R'$ and $S \mid T \xrightarrow{r\overline{\bowtie}\downarrow a} U$. This case is analogous to that of L-SESS-COM-PAR, the extra name restrictions in the resulting processes posing no additional problem.

∗ L-PAR-CLOSE': then $R \xrightarrow{r\bowtie\downarrow a} R'$ and $S \mid T \xrightarrow{r\overline{\bowtie}(a)\overline{a}} U$. This case is again analogous to the previous one.

The case when $Q \xrightarrow{\alpha} Q'$ is dealt with by a similar case analysis.

- *Name restriction*

– Parallel composition. Suppose P is $((\nu n)R) \mid S$ and Q is $(\nu n)(R \mid S)$. Assume first that $P \xrightarrow{\alpha} P'$; there are three different cases, according to which transition rule was used.

  * L-PAR: then $(\nu n)R \xrightarrow{\alpha} R'$. There are three possible sub-cases.

    · Suppose $(\nu n)R \xrightarrow{\alpha} R'$ follows by L-RES. Then n is not a name in α, R' is $(\nu n)R''$ and $R \xrightarrow{\alpha} R''$. Since n is also not a name in S, the following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{\alpha} R''}{R \mid S \xrightarrow{\alpha} R'' \mid S}\;\text{L-PAR}}{(\nu n)(R \mid S) \xrightarrow{\alpha} (\nu n)(R'' \mid S)}\;\text{L-RES}$$

    · Suppose $(\nu n)R \xrightarrow{\alpha} R'$ follows by L-SESS-RES. Then α is τ, R' is $(\nu n)R''$ and $R \xrightarrow{n\tau} R''$. Again, since n is also not a name in S, the following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{n\tau} R''}{R \mid S \xrightarrow{n\tau} R'' \mid S}\;\text{L-PAR}}{(\nu n)(R \mid S) \xrightarrow{\tau} (\nu n)(R'' \mid S)}\;\text{L-SESS-RES}$$

    · Suppose $(\nu n)R \xrightarrow{\alpha} R'$ follows by L-EXTR. Then α is $(n)\overline{n}$ and $R \xrightarrow{\overline{n}} R'$. Again, since n is also not a name in S, the following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{\overline{n}} R'}{R \mid S \xrightarrow{\overline{n}} R' \mid S}\;\text{L-PAR}}{(\nu n)(R \mid S) \xrightarrow{(n)\overline{n}} (R' \mid S)}\;\text{L-EXTR}$$

  In either case, it is easy to verify that $(\nu n)(R \mid S)$ evolves to a process structurally congruent to the evolution of $((\nu n)R) \mid S$.

  * L-PAR': then $S \xrightarrow{\alpha} S'$. Since $(\nu n)R \mid S$ is well-formed, n does not occur in S; therefore n cannot occur in α. Then the following derivation establishes the thesis.

$$\dfrac{\dfrac{S \xrightarrow{\alpha} S'}{R \mid S \xrightarrow{\alpha} R \mid S'}\;\text{L-PAR'}}{(\nu n)(R \mid S) \xrightarrow{\alpha} (\nu n)(R \mid S')}\;\text{L-RES}$$

  * L-SESS-COM-PAR: then α is $r\tau$, $(\nu n)R \xrightarrow{r\bowtie\Updownarrow v} R'$ and $S \xrightarrow{r\overline{\bowtie\Updownarrow} v} S'$. Then necessarily R' is $(\nu n)R''$ and the former transition is inferred via L-RES. The following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{r\bowtie\Updownarrow v} R'' \quad S \xrightarrow{r\overline{\bowtie\Updownarrow} v} S'}{R \mid S \xrightarrow{r\tau} R'' \mid S'}\;\text{L-SESS-COM-PAR}}{(\nu n)(R \mid S) \xrightarrow{r\tau} (\nu n)(R'' \mid S')}\;\text{L-RES}$$

  The cases when the rule applied is L-SERV-COM-PAR, L-PAR-CLOSE or L-PAR-CLOSE' are similar, except that further applications of S-SWAP may be necessary to verify that both processes evolve to structurally congruent processes.

Assume now that $Q \xrightarrow{\alpha} Q'$. Since the top-level constructor in Q is name restriction, there are three possible cases.

  * Assume the last rule applied is L-RES. Then $R \mid S \xrightarrow{\alpha} T$, with Q' being $(\nu n)T$ and n a name not occurring in α. There are six sub-cases, corresponding to the six different rules that may be used to infer the transition of $R \mid S$.

- L-PAR: then T is $R' \mid S$ with $R \xrightarrow{\alpha} R'$; then the following proof establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{\alpha} R'}{(\nu n)R \xrightarrow{\alpha} (\nu n)R'}\ \text{L-RES}}{((\nu n)R) \mid S \xrightarrow{\alpha} ((\nu n)R') \mid S}\ \text{L-PAR}$$

- L-PAR': then T is $R \mid S'$ with $S \xrightarrow{\alpha} S'$; the following proof establishes the thesis.

$$\dfrac{S \xrightarrow{\alpha} S'}{((\nu n)R) \mid S \xrightarrow{\alpha} ((\nu n)R) \mid S'}\ \text{L-PAR'}$$

- L-SESS-COM-PAR: then α is $r\tau$, $R \xrightarrow{r\bowtie\Updownarrow v} R'$, $S \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} S'$ and T is $R' \mid S'$. Care must be taken to distinguish whether n is v.
  If n is not v, then the following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{r\bowtie\Updownarrow v} R'}{(\nu n)R \xrightarrow{r\bowtie\Updownarrow v} (\nu n)R'}\ \text{L-RES} \qquad S \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} S'}{((\nu n)R) \mid S \xrightarrow{r\tau} ((\nu n)R') \mid S'}\ \text{L-SESS-COM-PAR}$$

  If n is v, then by well-formedness the process performing the output must be R (otherwise S would contain a binder n, which violates the assumption that all bound names in $R \mid S$ are distinct); the following proof establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{r\bowtie\Updownarrow v} R'}{(\nu n)R \xrightarrow{r\bowtie(n)\Updownarrow v} R'}\ \text{L-EXTR} \qquad S \xrightarrow{r\overline{\bowtie}\overline{\Updownarrow} v} S'}{((\nu n)R) \mid S \xrightarrow{r\tau} (\nu n)(R' \mid S')}\ \text{L-SESS-CLOSE}$$

- L-SERV-COM-PAR: then α is τ, $R \xrightarrow{a\Leftrightarrow(r)} R'$, $S \xrightarrow{a\overline{\Leftrightarrow}(r)} S'$ and T is $(\nu r)(R' \mid S')$. Notice that from the hypothesis it follows that n is distinct from r. Consider the following derivation.

$$\dfrac{\dfrac{R \xrightarrow{a\Leftrightarrow(r)} R'}{(\nu n)R \xrightarrow{a\Leftrightarrow(r)} (\nu n)R'}\ \text{L-RES} \qquad S \xrightarrow{a\overline{\Leftrightarrow}(r)} S'}{((\nu n)R) \mid S \xrightarrow{\tau} (\nu r)(((\nu n)R') \mid S')}\ \text{L-SERV-COM-PAR}$$

  Finally, using rules S-EXTR-PAR and S-SWAP, it follows that $(\nu r)(((\nu n)R') \mid S') \equiv (\nu r)(\nu n)(R' \mid S') \equiv Q'$.

- L-PAR-CLOSE: then α is $r\tau$, $R \xrightarrow{r\bowtie(a)\overline{a}} R'$, $S \xrightarrow{r\overline{\bowtie}\downarrow a} S'$ and T is $(\nu a)(R' \mid S')$. Again, by well-formedness, a is distinct from n. The following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{r\bowtie(a)\overline{a}} R'}{(\nu n)R \xrightarrow{r\bowtie(a)\overline{a}} (\nu n)R'}\ \text{L-RES} \qquad S \xrightarrow{r\overline{\bowtie}\downarrow a} S'}{((\nu n)R) \mid S \xrightarrow{r\tau} (\nu a)(((\nu n)R') \mid S')}\ \text{L-SESS-CLOSE}$$

- L-PAR-CLOSE': then α is $r\tau$, $R \xrightarrow{r\bowtie\downarrow a} R'$, $S \xrightarrow{r\overline{\bowtie}(a)\overline{a}} S'$ and T is $(\nu a)(R' \mid S')$. Again, by well-formedness, a is distinct from n. The following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{r\bowtie\downarrow a} R'}{(\nu n)R \xrightarrow{r\bowtie\downarrow a} (\nu n)R'}\ \text{L-RES} \qquad S \xrightarrow{r\overline{\bowtie}(a)\overline{a}} S'}{((\nu n)R) \mid S \xrightarrow{r\tau} (\nu a)(((\nu n)R') \mid S')}\ \text{L-SESS-CLOSE'}$$

* Assume the last rule applied is L-SESS-RES. This case is very similar to the previous one, but simpler: since session names may not be communicated, there are less possible cases and no need arises to use close rules.
* Assume the last rule applied is L-EXTR. Then α is $(n)\mu$, where μ is an output (session or stream). By well-formedness, n does not occur in S, whence it follows that necessarily $R \xrightarrow{\mu} R'$ and Q' is $R' \mid S$. Then the following proof shows that $P \xrightarrow{\alpha} Q'$.

$$\dfrac{\dfrac{R \xrightarrow{\mu} R'}{(\nu n)R \xrightarrow{(n)\mu} R'}\ \text{L-EXTR}}{((\nu n)R) \mid S \xrightarrow{(n)\mu} R' \mid S}\ \text{L-PAR}$$

- Composition with stream. There are two congruence rules for this case; both of them require a case analysis that is completely similar to that in the previous case (since composition with a stream is very similar to parallel composition). The extra case arising from L-FEED-CLOSE is similar to the other close rules.
- Session. Assume P is $r \bowtie ((\nu a)R)$ and Q is $(\nu a)(r \bowtie R)$. Suppose first that $P \xrightarrow{\alpha} P'$; there are two different cases.
  * L-SESS-VAL: then α is $r \bowtie \mu$, where μ is an input/output action. There are two possible sub-cases, according to how the transition of $(\nu a)R$ is inferred (since L-SESS-RES does not apply).
    · L-RES: then P' is $r \bowtie ((\nu a)R')$ with $R \xrightarrow{\mu} R'$, and the following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{\mu} R'}{r \bowtie R \xrightarrow{r\bowtie\mu} r \bowtie R'}\ \text{L-SESS-VAL}}{(\nu a)(r \bowtie R) \xrightarrow{r\bowtie\mu} (\nu a)(r \bowtie R')}\ \text{L-RES}$$

    · L-EXTR: then P' is $r \bowtie R'$, μ is $\overline{a}$, and the following derivation establishes the thesis.

$$\dfrac{\dfrac{R \xrightarrow{\overline{a}} R'}{r \bowtie R \xrightarrow{r\bowtie\overline{a}} r \bowtie R'}\ \text{L-SESS-VAL}}{(\nu a)(r \bowtie R) \xrightarrow{r\bowtie(a)\overline{a}} r \bowtie R'}\ \text{L-EXTR}$$

  * L-SESS-PASS: this case is very similar with only two differences. In the case of L-EXTR, μ is now $\Uparrow a$, and the rest follows as before. There is also the extra case of L-SESS-RES, which is straightforward.

  Assume now that $Q \xrightarrow{\alpha} Q'$. The proof is very similar, so we will only sketch it; there are three cases.
  * L-RES: then Q' is $(\nu a)S$ with $r \bowtie R \xrightarrow{\alpha} S$. There are two cases for the latter transition; in either of them, S must be of the form $r \bowtie R'$ and the thesis follows by swapping the application of the two rules.
  * L-SESS-RES: similar, but now there is only one sub-case, corresponding to L-SESS-PASS.
  * L-EXTR: then $r \bowtie R \xrightarrow{\mu} Q'$ and either α is $(a)\mu$ or α is $s \bowtie (a)\overline{a}$ and μ is $s \bowtie a$ for some session name s. Again there are two cases for the latter transition, and a straightforward swapping of the two rules yields the proof that $P \xrightarrow{\alpha} Q'$.
- Commutativity. Straightforward, since two different names are involved and well-formedness of the processes guarantees that all side conditions in the relevant rules will hold.
- Zero. Straightforward, since $(\nu a)0 \not\to$ and $0 \not\to$.

• *Recursion* This case is completely straightforward: if $\textbf{rec}\, X.R \xrightarrow{\alpha} P'$, then the only rule that can have been used to infer that transition is L-REC, whence it immediately follows that $R\left[{}^{\textbf{rec}\, X.R}/_{X}\right] \xrightarrow{\alpha} P'$. Reciprocally, if the latter condition holds, then by L-REC also $\textbf{rec}\, X.R \xrightarrow{\alpha} P'$.

□

C.2 Derivability of the new transition rules in the original LTS

Lemma 11. *Rule* L-PAR’ *is admissible in the original LTS for* SSCC.

Proof. The following derivation shows that any instance of L-PAR’ can be derived in the original LTS.

$$\dfrac{\dfrac{Q \xrightarrow{\mu} Q' \qquad \mathsf{bn}(\mu \cap \mathsf{fn}(P)) = \emptyset}{Q \mid P \xrightarrow{\mu} Q' \mid P}\ \text{L-PAR} \qquad Q \mid P \equiv P \mid Q \qquad Q' \mid P \equiv P \mid Q'}{P \mid Q \xrightarrow{\mu} P \mid Q'}\ \text{L-STRUCT}$$

□

Lemma 12. *Rule* L-REC *is admissible in the original LTS for* SSCC.

Proof. The following derivation shows that any instance of L-REC can be derived in the original LTS.

$$\dfrac{P\left[{}^{\mathsf{rec}\ X.P}/_{X}\right] \xrightarrow{\mu} P' \qquad P\left[{}^{\mathsf{rec}\ X.P}/_{X}\right] \equiv \mathsf{rec}\ X.P \qquad P' \equiv P'}{\mathsf{rec}\ X.P \xrightarrow{\mu} P'}\ \text{L-STRUCT}$$

□

The following lemma shows a property of transitions derived in the new LTS.

Lemma 13. *Let P and P' be processes and a be a name.*

1. *If $P \xrightarrow{(a)\uparrow a} P'$ then $P \equiv (\nu a)R$ for some R such that $R \xrightarrow{\uparrow a} P'$.*
2. *If $P \xrightarrow{r \bowtie (a)\uparrow a} P'$ then $P \equiv (\nu a)R$ for some R such that $R \xrightarrow{r \bowtie \uparrow a} P'$.*
3. *If $P \xrightarrow{(a)\Uparrow a} P'$ then $P \equiv (\nu a)R$ for some R such that $R \xrightarrow{\Uparrow a} P'$.*

Proof. All three parts of the lemma are proved by induction on the proof of the transition.

For (i), the base case is when rule L-EXTR is applied. Then the thesis follows immediately from the premise of the rule and reflexivity of $\equiv$. The induction cases are when one out of L-PAR, L-PAR’, L-STREAM-PASS-P, L-STREAM-PASS-Q, L-RES or L-REC is applied.

The first four cases are analogous. Suppose rule L-PAR was applied; then P is $P_1 \mid P_2$, P' is $P_1' \mid P_2$, $P_1 \xrightarrow{(a)\uparrow a} P_1'$ and a is not a free name of P_2. By induction hypothesis, $P_1 \equiv (\nu a)R$ with $R \xrightarrow{\uparrow a} P_1'$; also $P_1 \mid P_2 \equiv (\nu a)R \mid P_2$. By rule L-PAR, $R \mid P_2 \xrightarrow{\uparrow a} P'$. The case of L-REC is also straightforward: since $\mathsf{rec}\ X.P \equiv P\left[{}^{\mathsf{rec}\ X.P}/_{X}\right]$, the induction hypothesis immediately establishes the result. Finally, for L-RES, simply apply the induction hypothesis and use S-SWAP to conclude the thesis.

The proof of (ii) is completely similar except for the base case. Here, the rule being applied may also be L-SESS-VAL, in which case P is $r \bowtie Q$ and P' is $r \bowtie Q'$. By (i), also $Q \equiv (\nu a)R$ with $R \xrightarrow{\uparrow a} Q'$, whence $r \bowtie R \xrightarrow{r \bowtie \uparrow a} P'$. Since $r \bowtie (\nu a)R \equiv (\nu a) r \bowtie R$, the thesis follows.

The last case is analogous to the first. □

Lemma 14. *Rules* L-PAR-CLOSE *and* L-PAR-CLOSE’ *are admissible in the original LTS for* SSCC.

Proof. Suppose $P \mid Q \xrightarrow{r\tau} (\nu a)P' \mid Q'$ by L-PAR-CLOSE. By part (ii) of Lemma 13, $P \equiv (\nu a)R$ for some R such that $R \xrightarrow{r \bowtie \uparrow a} P'$. The following derivation shows that this instance of L-PAR-CLOSE can be derived in the original LTS.

$$\dfrac{\dfrac{\dfrac{R \xrightarrow{r \bowtie \uparrow a} P' \quad Q \xrightarrow{r \bowtie \downarrow a} Q'}{R \mid Q \xrightarrow{r\tau} P' \mid Q'}\ \text{L-PAR}}{(\nu a)R \mid Q \xrightarrow{r\tau} (\nu a)P' \mid Q'}\ \text{L-RES} \qquad (\nu a)R \equiv P \quad Q \equiv Q}{P \mid Q \xrightarrow{r\tau} (\nu a)P' \mid Q'}\ \text{L-STRUCT}$$

For rule L-PAR-CLOSE’, apply the previous construction with L-PAR’ instead of L-PAR and invoke Lemma 11. □

Lemma 15. *Rules* L-SESS-CLOSE *and* L-SESS-CLOSE’ *are admissible in the original LTS for* SSCC.

Proof. Analogous to the previous one. □

Lemma 16. *Rule* L-FEED-CLOSE *is admissible in the original LTS for* SSCC.

Proof. Suppose $\textsf{\textbf{stream}}\, P\, \textsf{\textbf{as}}\, f = \vec{w}\, \textsf{\textbf{in}}\, Q \xrightarrow{\tau} (\nu a)\textsf{\textbf{stream}}\, P'\, \textsf{\textbf{as}}\, f = a :: \vec{w}\, \textsf{\textbf{in}}\, Q$ by L-FEED-CLOSE. By part (iii) of Lemma 13, $P \equiv (\nu a)R$ for some R such that $R \xrightarrow{\Uparrow a} P'$. The following derivation shows that this instance of L-PAR-CLOSE can be derived in the original LTS. Note, in fact, that the application of L-STRUCT is sound, since from $(\nu a)R \equiv P$ it follows that $(\nu a)\textsf{\textbf{stream}}\, R\, \textsf{\textbf{as}}\, f = \vec{w}\, \textsf{\textbf{in}}\, Q \equiv \textsf{\textbf{stream}}\, P\, \textsf{\textbf{as}}\, f = \vec{w}\, \textsf{\textbf{in}}\, Q$ (for the left-hand-side), and since $\equiv$ is reflexive (for the right-hand-side).

$$\cfrac{\cfrac{\cfrac{R \xrightarrow{\Uparrow a} P'}{\textsf{\textbf{stream}}\, R\, \textsf{\textbf{as}}\, f = \vec{w}\, \textsf{\textbf{in}}\, Q \xrightarrow{\tau} \textsf{\textbf{stream}}\, P'\, \textsf{\textbf{as}}\, f = a :: \vec{w}\, \textsf{\textbf{in}}\, Q}\ \text{L-STREAM-FEED}}{(\nu a)\textsf{\textbf{stream}}\, R\, \textsf{\textbf{as}}\, f = \vec{w}\, \textsf{\textbf{in}}\, Q \xrightarrow{\tau} (\nu a)\textsf{\textbf{stream}}\, P'\, \textsf{\textbf{as}}\, f = a :: \vec{w}\, \textsf{\textbf{in}}\, Q}\ \text{L-RES}}{\textsf{\textbf{stream}}\, P\, \textsf{\textbf{as}}\, f = \vec{w}\, \textsf{\textbf{in}}\, Q \xrightarrow{\tau} (\nu a)\textsf{\textbf{stream}}\, P'\, \textsf{\textbf{as}}\, f = a :: \vec{w}\, \textsf{\textbf{in}}\, Q}\ \text{L-STRUCT}$$

□

D On the bisimilarities of SSCC

D.1 Strong bisimilarity

We study here strong bisimilarity, hereafter referred to simply as “bisimilarity”, as defined in Definition 10. Remember that bisimilarity can be obtained as the union of all bisimulations or as a fixed-point of a suitable monotonic operator; also it is well defined, as the next result shows.

Theorem 6. *Structurally congruent processes are bisimilar.*

Proof. It suffices to show that $\equiv$ is a bisimulation, which is an immediate consequence of the Harmony Lemma. □

We now show that bisimilarity is a non-input congruence, just as in π-calculus. The strategy of the proof is the same as in [41], based on the notion and properties of a relation *progressing* to another relation.

Definition 17. *A relation $\mathcal{R}$ on processes* strongly progresses *to another relation $\mathcal{S}$, denoted $\mathcal{R} \rightsquigarrow \mathcal{S}$, if, whenever $P\mathcal{R}Q$, $P \xrightarrow{\alpha} P'$ implies $Q \xrightarrow{\alpha} Q'$ for some Q' with $P'\mathcal{S}Q'$, and vice-versa.*

Definition 18. *A function $\mathcal{F}$ on processes is* strongly safe *if $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightsquigarrow \mathcal{S}$ imply $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightsquigarrow \mathcal{F}(\mathcal{S})$.*

Lemma 17. *If $\mathcal{F}$ is strongly safe and $\sim \subseteq \mathcal{F}(\sim)$, then $F(\sim) = \sim$.*

Proof. See [41]. □

Given a function $\mathcal{F}$, define $\mathcal{F}^*$ such that $\mathcal{F}^*(\mathcal{R})$ is the transitive closure of $\mathcal{F}(\mathcal{R})$.

Lemma 18. *If $\mathcal{F}$ is such that $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightsquigarrow \mathcal{S}$ imply that $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}^*(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightsquigarrow \mathcal{F}^*(\mathcal{S})$, then $\mathcal{F}^*$ is strongly safe.*

Proof. See [41]. □

The proof relies on defining functions $\mathcal{F}_{\text{ni1}}$ and $\mathcal{F}_{\text{ni}}$ like those for π-calculus; however, the definition of the former has to be slightly adapted.

Definition 19.

- *An n-ary* multi-hole context *C is a process where some occurrences of* 0 *have been replaced by holes $[\cdot]_i$; each hole may occur zero or more times. Given n processes $P_1, \ldots, P_n$, $C[P_1, \ldots, P_n]$ is the process obtained by uniformly replacing all occurrences of all holes in C by the corresponding process.*
- *A (multi-hole) context is said to be* non-input *if no hole occurs under an input prefix (x) or $f(x)$.*
- *Functions $\mathcal{F}_{ni1}$ and $\mathcal{F}_{ni}$ are defined as follows.*

$$\begin{aligned}\mathcal{F}_{ni1}(\mathcal{R}) &= \{\langle C[P], C[Q]\rangle \mathbin{[\!]} P\mathcal{R}Q \text{ and } C \text{ is a non-input context}\}\\ \mathcal{F}_{ni}(\mathcal{R}) &= \{\langle C[P_1, \ldots, P_n], C[Q_1, \ldots, Q_n]\rangle \mathbin{[\!]} P_i\mathcal{R}Q_i \text{ and } C \text{ is an } n\text{-ary non-input context}\}\end{aligned}$$

Lemma 19. $\mathcal{F}_{ni} = \mathcal{F}_{ni1}^*$.

Proof. As for π-calculus. □

Lemma 20. *Function $\mathcal{F}_{ni}$ is strongly safe.*

Proof. Applying Lemma 18, one must show that, whenever $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightsquigarrow \mathcal{S}$, both $\mathcal{F}_{\text{ni1}}(\mathcal{R}) \subseteq \mathcal{F}_{\text{ni1}}(\mathcal{S})$ and $\mathcal{F}_{\text{ni1}}(\mathcal{R}) \rightsquigarrow \mathcal{F}_{\text{ni}}(\mathcal{S})$. The first of these is trivial by definition of $\mathcal{F}_{\text{ni1}}$.

Assume that $P\mathcal{R}Q$. One must show that, for every context C, if $C[P] \xrightarrow{\alpha} P'$, then $C[Q] \xrightarrow{\alpha} Q'$ for some P' and Q' such that there exist an n-ary context C' and processes $P_1\mathcal{S}Q_1, \ldots, P_n\mathcal{S}Q_n$ for which P' is $C'[P_1, \ldots, P_n]$ and Q' is $C'[Q_1, \ldots, Q_n]$.

The proof is by induction on the derivation tree for $C[P] \xrightarrow{\alpha} P'$. In all steps, there are two cases to consider, according to whether C is $[\cdot]$ or not; the former case is always trivial, since the hypothesis $\mathcal{R} \rightsquigarrow \mathcal{S}$ establishes the thesis. Therefore, we always assume below that C is not $[\cdot]$. The proof looks at the last rule being applied.

- L-SEND: then C is $v.C_0$ and α is $\uparrow v$ for some v. Furthermore, $v.C_0[Q] \xrightarrow{\uparrow v} C_0[Q]$; since C_0 is also a multi-hole context and $\mathcal{R} \subseteq \mathcal{S}$, it follows that $\langle C_0[P], C_0[Q]\rangle \in \mathcal{F}_{\text{ni}}(\mathcal{S})$, hence the thesis holds.
- L-RECEIVE: then C is $(x)C_0$, and since C is a non-input context (by definition of $\mathcal{F}_{\text{ni1}}$), it follows that C_0 does not contain holes; hence in this case $C[P]$ and $C[Q]$ coincide, and the result is trivial.
- L-FEED: then C is **feed** $v.C_0$ and α is $\Uparrow v$ for some v. Furthermore, **feed** $v.C_0[Q] \xrightarrow{\Uparrow v} C_0[Q]$; since C_0 is also a multi-hole context and $\mathcal{R} \subseteq \mathcal{S}$, it follows that $\langle C_0[P], C_0[Q]\rangle \in \mathcal{F}_{\text{ni}}(\mathcal{S})$, hence the thesis holds.
- L-READ: then C is $f(x).C_0$, and since C is a non-input context (by definition of $\mathcal{F}_{\text{ni1}}$), it follows that C_0 does not contain holes; hence in this case $C[P]$ and $C[Q]$ coincide, and the result is trivial.
- L-CALL: then C is $a \Leftarrow C_0$ and α is $a \Leftarrow (r)$ for some r not occurring free in $C_0[P]$. Furthermore, $a \Leftarrow C_0[Q] \xrightarrow{a \Leftarrow (r)} r \lhd C_0[Q]$, since by definition of bisimulation r does not occur free in $C_0[Q]$. Taking C' to be the context $r \lhd C_0$ establishes the thesis.
- L-INV: analogous.
- L-PAR: there are two cases to consider.
  - If C is $C_0 \mid R$, then $C_0[P] \xrightarrow{\alpha} P'$ and α and R share no bound names. By induction hypothesis there exists a process Q' such that $C_0[Q] \xrightarrow{\alpha} Q'$, and P', Q' are respectively $C_0'[P_1, \ldots, P_n]$ and $C_0'[Q_1, \ldots, Q_n]$ for some n-ary multi-hole context C_0' and processes $P_1\mathcal{S}Q_1, \ldots, P_n\mathcal{S}Q_n$. Thus $C_0[Q] \mid R \xrightarrow{\alpha} Q' \mid R$, hence taking C' to be $C_0' \mid R$ establishes the thesis.

- If C is $R \mid C_0$, then $R \xrightarrow{\alpha} R'$ and $C_0[P]$ and R share no bound names. By the hypothesis of L-PAR, $C_0[Q]$ and R also share no bound names, hence $R \mid C_0[Q] \xrightarrow{\alpha} R \mid Q'$, and taking C' to be $R \mid C'_0$ establishes the thesis.

- L-PAR': analogous (the two cases are reversed).
- L-STREAM-PASS-P and L-STREAM-PASS-Q: analogous to L-PAR and L-PAR', respectively.
- L-STREAM-FEED: there are two cases to consider.
  - If C is $\textbf{stream}\, C_0\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, R$, then $C_0[P] \xrightarrow{\Uparrow v} P'$; by induction hypothesis, there exists a process Q' such that $C_0[Q] \xrightarrow{\Uparrow v} Q'$, and P' and Q' are respectively $C'_0[P_1, \ldots, P_n]$ and $C'_0[Q_1, \ldots, Q_n]$ for some n-ary multi-hole context C'_0 and processes $P_1 \mathcal{S} Q_1, \ldots, P_n \mathcal{S} Q_n$. Thus $\textbf{stream}\, C_0[Q]\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, R \xrightarrow{\tau} \textbf{stream}\, Q'\, \textbf{as}\, f = v :: \vec{w}\, \textbf{in}\, R$, hence taking C' to be $\textbf{stream}\, C'_0\, \textbf{as}\, f = v :: \vec{w}\, \textbf{in}\, R$ establishes the thesis.
  - If C is $\textbf{stream}\, R\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, C_0$, then $R \xrightarrow{\Uparrow v} R'$, hence $\textbf{stream}\, R\, \textbf{as}\, f = \vec{w}\, \textbf{in}\, C_0[Q] \xrightarrow{\tau} \textbf{stream}\, R'\, \textbf{as}\, f = v :: \vec{w}\, \textbf{in}\, C_0[Q]$, hence taking C' to be $\textbf{stream}\, R'\, \textbf{as}\, f = v :: \vec{w}\, \textbf{in}\, C_0$ establishes the thesis.
- L-STREAM-CONS: analogous (the two cases are reversed).
- L-SESS-VAL: then C is $r \bowtie C_0$, α is $r \bowtie \mu$ for some μ, and $C_0[P] \xrightarrow{\mu} P_0$. By induction hypothesis, $C_0[Q] \xrightarrow{\mu} Q_0$ for some Q_0 such that there exist a multi-hole context C'_0 and processes $P_1 \mathcal{S} Q_1, \ldots, P_n \mathcal{S} Q_n$ for which P_0 is $C'_0[P_1, \ldots, P_n]$ and Q_0 is $C'_0[Q_1, \ldots, Q_n]$. Taking C' to be $r \bowtie C'_0$ establishes the thesis, since then $r \bowtie C_0[Q] \xrightarrow{r \bowtie \mu} C'[Q_1, \ldots, Q_n]$.
- L-SESS-PASS: then C is $r \bowtie C_0$, α is neither an input nor an output, and $C_0[P] \xrightarrow{\alpha} P_0$. The proof then follows as above except that the action does not change when the session is added to C'_0.
- L-SESS-COM-PAR: C is either $C_0 \mid R$ or $R \mid C_0$; the two cases are analogous, so assume the first holds. Then $C_0[P] \xrightarrow{r \bowtie \Updownarrow v} P'$, $R \xrightarrow{r \overline{\bowtie} \Updownarrow v} R'$, α is $r\tau$ for some r and $C_0[P] \mid R \xrightarrow{r\tau} P' \mid R'$. By induction hypothesis there exists a process Q' such that $C_0[Q] \xrightarrow{r \bowtie \Updownarrow v} Q'$, and P', Q' are respectively $C'_0[P_1, \ldots, P_n]$ and $C'_0[Q_1, \ldots, Q_n]$ for some n-ary multi-hole context C'_0 and processes $P_1 \mathcal{S} Q_1, \ldots, P_n \mathcal{S} Q_n$. Then $C_0[Q] \mid R \xrightarrow{r\tau} Q' \mid R'$, hence taking C' to be $C'_0 \mid R'$ establishes the thesis.
- the cases concerning rules L-SERV-COM-PAR, L-SESS-COM-STREAM, L-SERV-COM-STREAM, L-PAR-CLOSE, L-PAR-CLOSE', L-FEED-CLOSE, L-SESS-COM-CLOSE and L-SERV-COM-CLOSE are all very similar to the previous one.
- L-RES: then C is $(\nu a)C_0$, $(\nu a)C_0[P] \xrightarrow{\alpha} (\nu a)P'$, a is not a name in α and $C_0[P] \xrightarrow{\alpha} P'$. By induction hypothesis there exists a process Q' such that $C_0[Q] \xrightarrow{\alpha} Q'$, and P', Q' are respectively $C'_0[P_1, \ldots, P_n]$ and $C'_0[Q_1, \ldots, Q_n]$ for some n-ary multi-hole context C'_0 and processes $P_1 \mathcal{S} Q_1, \ldots, P_n \mathcal{S} Q_n$. Thus $(\nu a)C_0[Q] \xrightarrow{\alpha} (\nu a)Q'$, hence taking C' to be $(\nu a)C'_0$ establishes the thesis.
- L-SESS-RES: analogous, only now α is τ and the induction hypothesis is applied to a transition via $a\tau$.
- L-EXTR: analogous, only now α is an action extruding a and the induction hypothesis is applied to a transition without the extrusion; furthermore, the context C' is simply C'_0.
- L-REC: C is $\textbf{rec}\, X.C_0$ and $C_0[P] \left[{}^{\textbf{rec}\, X.C_0[P]}/_{X}\right] \xrightarrow{\alpha} P'$. Since P is well-formed, it contains no free occurrences of X; hence there exists a context C_1 such that $C_1[P]$ is $C_0[P] \left[{}^{\textbf{rec}\, X.C_0[P]}/_{X}\right]$ and $C_1[Q]$ is $C_0[Q] \left[{}^{\textbf{rec}\, X.C_0[Q]}/_{X}\right]$. Hence the induction hypothesis applies, and there exist a

process Q' and a context C' such that $C_1[Q] \xrightarrow{\alpha} Q'$, P' is $C'[P]$ and Q' is $C'[Q]$. Therefore also $\textbf{rec}\, X.C_0[Q] \xrightarrow{\alpha} Q'$, and C' is the required context.

□

Theorem 7. *Bisimilarity is a non-input congruence.*

Proof. Straightforward consequence of Lemmas 17 and 20. □

At this point we can explain in more detail why the original LTS for SSCC had to be changed. Consider any derivation containing an application of L-STRUCT.

$$\frac{P \xrightarrow{\mu} P' \qquad P \equiv Q \qquad P' \equiv Q'}{Q \xrightarrow{\mu} Q'} \text{ L-STRUCT}$$

In general, the induction hypothesis will not be applicable to the subtree that shows $P \xrightarrow{\mu} P'$, since there is no obvious relationship between P and Q; furthermore, the thesis of the induction hypothesis does not help in establishing the final result, since again there is no obvious relationship between Q' and P'.

Observe also that this is not a problem of this particular proof technique. Whether the induction were on the derivation tree (as above), on contexts (as the proof for π-calculus, see [41]) or on processes (arguably an alternative) the same problem would arise, since the issue arises from the fact that the theorem assumes hypotheses on the actual process performing the transition. This justifies the attempt to eliminate L-STRUCT from the LTS altogether.

D.2 Weak bisimilarity

We turn now to weak bisimilarity, defined according to Definition 12. Weak bisimilarity treats internal actions as irrelevant. Again, weak bisimilarity can be obtained as the union of all weak bisimulations or as a fixed-point of a suitable monotonic operator.

Theorem 8. *Let $\simeq$ be the largest relation such that, whenever $P \simeq Q$, for every process P' and action α, if $P \xrightarrow{\alpha} P'$, then $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$ for some Q' with $P' \simeq Q'$ and vice-versa. Then $P \approx Q$ iff $P \simeq Q$.*

Proof. The direct implication is straightforward, since $P \xrightarrow{\alpha} P'$ implies that $P \overset{\hat{\alpha}}{\Rightarrow} P'$. For the converse, assume that $P \overset{\hat{\alpha}}{\Rightarrow} P'$. If P' is P and α is τ, then result is trivial; otherwise, there exist processes $P_1, \ldots, P_n$ and $P'_1, \ldots, P'_m$ such that P is P_1, $P_i \xrightarrow{\tau} P_{i+1}$ for $i < n$, $P_n \xrightarrow{\alpha} P'_1$, $P'_j \xrightarrow{\tau} P'_{j+1}$ for $j < m$ and P'_m is P'. By hypothesis, there exist processes $Q_1, \ldots, Q_n$ and $Q'_1, \ldots, Q'_m$ (not necessarily distinct) such that $Q_i \overset{\hat{\tau}}{\Rightarrow} Q_{i+1}$ for $i < n$, $Q_n \overset{\hat{\alpha}}{\Rightarrow} Q'_1$ and $Q'_j \overset{\hat{\tau}}{\Rightarrow} Q'_{j+1}$ for $j < m$; furthermore, $P_i \simeq Q_i$ and $P'_j \simeq Q'_j$ for all $i \leq n$ and $j \leq m$. In particular, $Q \overset{\hat{\alpha}}{\Rightarrow} Q'_m$ and $P' \simeq Q'_m$, so $\simeq$ is a weak bisimulation. □

The reason for introducing $\simeq$ is that this relation is simpler to work with when proving properties by induction. In turn, the definition of $\approx$ is more symmetric and its relationship with $\sim$ is immediate.

We now show that bisimilarity is a non-input congruence, again like in π-calculus. The strategy of the proof is once more the same as in [41].

Definition 20. *A relation $\mathcal{R}$ on processes* progresses *to another relation $\mathcal{S}$, denoted $\mathcal{R} \rightsquigarrow \mathcal{S}$, if, whenever $P\mathcal{R}Q$, $P \xrightarrow{\alpha} P'$ implies $Q \overset{\hat{\alpha}}{\Rightarrow} Q'$ for some Q' with $P'\mathcal{S}Q'$, and vice-versa.*

Definition 21. *A function $\mathcal{F}$ on processes is* safe *if $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightsquigarrow \mathcal{S}$ imply $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightsquigarrow \mathcal{F}(\mathcal{S})$.*

Lemma 21. *If $\mathcal{F}$ is safe and $\approx \subseteq \mathcal{F}(\approx)$, then $F(\approx) = \approx$.*

Proof. See [41]. □

As is the case with π-calculus, proving that $\mathcal{F}_{\text{ni}}$ is safe must be done directly, since chaining is not secure.

Lemma 22. *Function $\mathcal{F}_{ni}$ is safe.*

Proof. Let $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightsquigarrow \mathcal{S}$. It must be shown that $\mathcal{F}_{\text{ni1}}(\mathcal{R}) \subseteq \mathcal{F}_{\text{ni1}}(\mathcal{S})$ and $\mathcal{F}_{\text{ni}}(\mathcal{R}) \rightsquigarrow \mathcal{F}_{\text{ni}}(\mathcal{S})$. As before, the first of these is trivial by definition of $\mathcal{F}_{\text{ni1}}$.

Assume that $P_i \mathcal{R} Q_i$ for $i = 1, \ldots, n$. It must be shown that, for every multi-context C, if there is a transition $C[P_1, \ldots, P_n] \xrightarrow{\alpha} P'$, then $C[Q_1, \ldots, Q_n] \overset{\alpha}{\Rightarrow} Q'$ for some P' and Q' such that there exist another multi-context C' and processes $P_1 \mathcal{S} Q_1, \ldots, P_m \mathcal{S} Q_m$ for which P' is $C'[P_1, \ldots, P_m]$ and Q' is $C'[Q_1, \ldots, Q_m]$.

Once more we proceed by induction on the derivation tree for $C[P_1, \ldots, P_n] \xrightarrow{\alpha} P'$. Most cases are very similar to the proof of Lemma 20; however, since the induction hypothesis now gives a weak transition some care must be taken.

As before, the case when C is $[\cdot]$ is straightforward; furthermore, the cases of rules L-SEND, L-RECEIVE, L-FEED, L-READ, L-CALL, L-INV, L-SESS-VAL, L-SESS-PASS, L-SESS-COM-PAR, L-SERV-COM-PAR, L-SESS-COM-STREAM, L-SERV-COM-STREAM,L-PAR-CLOSE, L-PAR-CLOSE', L-FEED-CLOSE, L-SESS-COM-CLOSE, L-SERV-COM-CLOSE, L-RES, L-SESS-RES, L-EXTR and L-REC are dealt with as in the proof of Lemma 20, with an extra step at the end (to take care of the possible extra τ steps) similar to the cases detailed above.

The only remaining cases are those when C is either a parallel composition or a stream composition, since now both subprocesses may be contexts.

- L-PAR: then C is $C_1 \mid C_2$, $C_1[P_1, \ldots, P_n] \xrightarrow{\alpha} P'$, and α and $C_2[P_1, \ldots, P_n]$ share no bound names. By induction hypothesis there exists a process Q' such that $C_1[Q_1, \ldots, Q_n] \overset{\alpha}{\Rightarrow} Q'$, and P', Q' are respectively $C'_1[P'_1, \ldots, P'_m]$ and $C'_1[Q'_1, \ldots, Q'_m]$ for some multi-hole context C'_1 and processes $P'_1 \mathcal{S} Q'_1, \ldots, P'_m \mathcal{S} Q'_m$. By applying L-PAR to all steps of the sequence of transitions $C_1[Q_1, \ldots, Q_n] \xrightarrow{\tau} \ldots \xrightarrow{\alpha} \ldots \xrightarrow{\tau} Q'$, we conclude that

$$C_1[Q_1, \ldots, Q_n] \mid C_2[Q_1, \ldots, Q_n] \overset{\alpha}{\Rightarrow} C'_1[Q'_1, \ldots, Q'_m] \mid C_2[Q_1, \ldots, Q_n],$$

  hence taking C' to be $C'_1 \mid C_2$ establishes the thesis[5].

- L-PAR': analogous (the roles of C_1 and C_2 are reversed).

- L-STREAM-PASS-P and L-STREAM-PASS-Q: as before, these are analogous to L-PAR and L-PAR', respectively.

- L-STREAM-FEED: then C is $\textsf{\textbf{stream}}\, C_1 \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, C_2$ and $C_1[P_1, \ldots, P_n] \xrightarrow{\Uparrow v} P'$. By induction hypothesis, there exists a process Q' such that $C_1[Q_1, \ldots, Q_n] \overset{\Uparrow v}{\Longrightarrow} Q'$, and P' and Q' are respectively $C'_1[P'_1, \ldots, P'_n]$ and $C'_1[Q'_1, \ldots, Q'_n]$ for some n-ary multi-hole context C'_1 and processes $P'_1 \mathcal{S} Q'_1, \ldots, P'_n \mathcal{S} Q'_n$. In other words, $C_1[Q_1, \ldots, Q_n] \overset{\tau}{\Rightarrow} Q^* \xrightarrow{\Uparrow v} Q^{**} \overset{\tau}{\Rightarrow} Q'$; using L-STREAM-PASS-P for the τ transitions, we conclude that

$$\begin{aligned}
&\textsf{\textbf{stream}}\, C_1[Q_1, \ldots, Q_n] \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, C_2[Q_1, \ldots, Q_n] \\
&\quad \overset{\tau}{\Rightarrow} \textsf{\textbf{stream}}\, Q^* \,\textsf{\textbf{as}}\, f = \vec{w} \,\textsf{\textbf{in}}\, C_2[Q_1, \ldots, Q_n] \\
&\quad \xrightarrow{\tau} \textsf{\textbf{stream}}\, Q^{**} \,\textsf{\textbf{as}}\, f = v :: \vec{w} \,\textsf{\textbf{in}}\, C_2[Q_1, \ldots, Q_n] \\
&\quad \overset{\tau}{\Rightarrow} \textsf{\textbf{stream}}\, C'_1[Q'_1, \ldots, Q'_n] \,\textsf{\textbf{as}}\, f = v :: \vec{w} \,\textsf{\textbf{in}}\, C_2[Q_1, \ldots, Q_n],
\end{aligned}$$

  hence taking C' to be $\textsf{\textbf{stream}}\, C'_1 \,\textsf{\textbf{as}}\, f = v :: \vec{w} \,\textsf{\textbf{in}}\, C_2$ establishes the thesis.

- L-STREAM-CONS: analogous (the roles of C_1 and C_2 are reversed).

□

Theorem 9. *Weak bisimilarity is a non-input congruence.*

Proof. Straightforward consequence of Lemmas 21 and 22. □

[5]We assume that a n-hole context does not have to contain occurrences of all its holes, so in particular C_1 and C_2 are n-hole contexts in which some holes may not occur.