Foundations of Session Types and Behavioural Contracts

Hans Hüttel, Emilio Tuosto, Hugo Torres Vieira, Gianluigi Zavattaro, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, et al.

Foundations of Session Types and Behavioural Contracts

Hans Hüttel, Aalborg University
Ivan Lanese, University of Bologna/INRIA
Vasco T. Vasconcelos, University of Lisbon
Luís Caires, Universidade Nova de Lisboa
Marco Carbone, ITU Copenhagen
Pierre-Malo Deniélou, Royal Holloway, University of London
Dimitris Mostrous, University of Lisbon
Luca Padovani, University of Turin
António Ravara, Universidade Nova de Lisboa
Emilio Tuosto, University of Leicester
Hugo Torres Vieira, IMT School for Advanced Studies Lucca
Gianluigi Zavattaro, University of Bologna/INRIA

Behavioural type systems, usually associated to concurrent or distributed computations, encompass concepts such as interfaces, communication protocols, and contracts, in addition to the traditional input/output operations. The behavioural type of a software component specifies its expected patterns of interaction using expressive type languages, so that types can be used to determine automatically whether the component interacts correctly with other components. Two related important notions of behavioural types are those of session types and behavioural contracts. This paper surveys the main accomplishments of the last twenty years within these two approaches.

Categories and Subject Descriptors: F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Theory, Languages, Verification

Additional Key Words and Phrases: Behavioural types

ACM Reference Format:
Hüttel et al.: Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* V, N, Article A (January YYYY), 36 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Types make it possible to classify entities of a program and to describe the permissible results of a computation. A type discipline can guarantee that well-typed programs are well-behaved. Traditionally the focus of the work on type systems has been on the outcome of computations, that is, on *what* the result of a computation should be.

During the 1990's, program semantics, in particular concurrency theory and especially the study of type disciplines for process calculi, has inspired notions of typing that are also able to describe properties associated with the behaviour of programs and in this way also describe *how* a computation proceeds. This often includes accounting for notions such as causality, choice and resource usage. Type disciplines that describe such notions directly are often referred to as *behavioural types*.

There is no hard and fast line of demarcation between behavioural type systems and other type systems. The work on behavioural types arose in the context of type systems that capture properties of computations in process calculi. While these systems do not describe the behavioural information directly as part of the type language, some of them have been instrumental in the development of the behavioural type systems presented in the current paper, e.g., by introducing notions of separation between *capabilities* for names [Pierce and Sangiorgi 1996; Kobayashi 2003], by considering *linear*

usage of names [Kobayashi et al. 1999; Kobayashi 2003], and by making case analyses of *variant* types [Sangiorgi 1998].

Like many other type disciplines, most of the approaches to behavioural type systems are compositional in the sense that the type of a composite program depends on the types of its immediate constituents.

Two notions of behavioural types that have attracted interest are those of session types and behavioural contracts, and in this paper we provide a survey of the most relevant work on the foundations of these two notions of behavioural type and outline the relation to other notions.

The structure of the paper is as follows. Section 2 sketches approaches to behavioural types that are related to those of session types and behavioural contracts. Section 3 describes session types and behavioural contracts for sessions involving two participants only, called binary sessions. Section 4 goes on to describe notions of session types and behavioural contracts that focus on multiparty interaction. Section 5 describes approaches to subtyping, refinement and polymorphism in these settings.

The remaining sections describe the expressiveness and algorithmic properties of session types and behavioural contracts. Section 6 outlines what is known about the relationship between session types and logic. Section 7 show how safety and liveness properties can be addressed using behavioural types. Section 8 describes how the various approaches to behavioural types interrelate. Finally, Section 9 deals with algorithmic properties, including decidability results for typing and subtyping.

2. OTHER APPROACHES TO BEHAVIOURAL TYPES

This section briefly presents approaches to behavioural types that relate to session types and behavioural contracts. Some of these have been important sources of inspiration in the development of the theory of session types and behavioural contracts. In what follows we shall restrict our attention to the trends which had more impact on the development of session types and behavioural contracts.

2.1. Intersection types

An intersection type system introduces a type constructor $\wedge$; an entity has type $T_1 \wedge T_2$ if it has both type T_1 and type T_2. This makes it possible to type a program that can exhibit behaviour corresponding to T_1 as well as behaviour corresponding to T_2, thereby enabling a notion of (ad-hoc) polymorphism.

Intersection types first arose in the setting of typed λ-calculi [Barendregt et al. 1983] and are closely related to the model theory of λ-calculus [Barendregt et al. 2013]. In some intersection type systems, typability characterises normalisation behaviour of terms in the λ-calculus (including exact characterisation of the strongly normalising terms [Pottinger 1980; Dezani-Ciancaglini et al. 2005]). Intersection types have first been integrated into a programming language by Reynolds [1997]. They have also been used to express behavioural abstractions of program behaviour in settings including abstract interpretation [Coppo and Ferrari 1993], type refinement [Freeman and Pfenning 1991], model checking [Naik and Palsberg 2005; Kobayashi and Ong 2009], and synthesis [Rehof 2013]. As a consequence, intersection type systems can be regarded as premier examples of behavioural type systems.

The dual notion of *union types* has often been introduced alongside intersection types [Pierce 1991; Barbanera et al. 1995; Dunfield and Pfenning 2003; Igarashi and Nagira 2007; Dunfield 2012]. Union types are used to express the *uncertainty* as to which type an entity has. For instance, a union type $\mathsf{Int} \vee \mathsf{String}$ describes a value that can be either an Int or a String and a program using such a value must account for both possibilities.

It is interesting to note the connection between more traditional type theory and process-oriented behavioural types. Bettini et al. [2008] combine session types

and union types while Padovani [2010b] traces a correspondence between intersection/union types and selection/branching constructs of binary session types.

2.2. Typestates

Typestates are a notion of behavioural types dating back to Strom and Yemini [1986]. In this approach, the type of an entity depends on the operations that are permitted for the entity, when at a particular state. Each type has associated with it a set of typestates, partially ordered; operations on entities of the type are correct if the resulting values are of a typestate reachable by a typestate transition (following the order).

Therefore, typestates are akin to finite-state machines, and a language equipped with a static type system based on them can check at compile time if all possible sequences of operations are valid with respect to a correct use of the application.

The original work on typestates considered imperative languages without objects but the notion has since been taken up by as a behavioural type discipline for object-oriented programming languages. DeLine and Fähndrich [2004] describe a programming language called Fugue, while Sunshine et al. [2011] have developed the Plaid programming language. In Plaid typestates incorporate into the traditional notion of class type (the interface, or the collection of method signatures) the representation (the fields) and the behaviour (the actual implementations of methods). Typestates may change over time, and the type system of Plaid makes it possible to track these changes. Gay et al. [2010] give semantics to a distributed concurrent object-oriented programming language by means of a unified treatment of communication channels and their session types (see Section 3) together with a notion of typestates that supports non-uniform objects (see Section 2.4).

2.3. Types and effects

The first proposal of behavioural types for concurrency was, to our knowledge, made by Nielson and Nielson [1993] in the setting of the concurrent functional language Concurrent ML, and then extended in [Nielson and Nielson 1996]. The authors develop a *type and effect discipline*. A type and effect system makes it possible to statically describe intensional aspects of a computation alongside the extensional information that is captured by usual notions of type. The distinction is that the type describes *what* an expression will compute (sets of values), while the effect describes *how* an expression will compute (the behaviour). In this approach, type judgements for programs P are of the form $\Gamma \vdash P\colon T\&B$. Here Γ denotes a type environment recording the types of free variables, T denotes the type of P and B the effect associated with executing P. The intent is *not* to reject a well-typed program based on the shape of its effects but to provide an upper bound on the actual effect that will be exhibited by P during its computation.

In a polymorphic functional language, a type and effect system can be used to control resource usage, such as memory management. When the programming language includes the notions of communication and concurrency, effects (also called *behaviours*) are terms of a process algebra and can, just like the programming language itself, be given a labelled transition semantics [Nielson et al. 1999]. A main feature of these type systems is that whenever a well-typed program performs a communication c, the effect of the residual of the program is the effect of the entire program minus the effect corresponding to the communication c. Note that session types evolve in a similar way, changing as far as the computation progresses and communications are performed.

2.4. Types for non-uniform objects

In the context of object-oriented programming, Nierstrasz [1995] observed that class types as static interface types did not cope with the notion of *non-uniform method*

availability: in an object, each of its methods can be enabled or disabled according to its internal state. A simple example is that of a queue (the dequeue method is disabled if the queue is empty), another is that of a finite buffer (here the write method is disabled if the buffer is full). *Non-uniform objects* are those that may dynamically change behaviour, and a typing discipline for ensuring the absence of 'message-not-understood' errors will need to take this dynamic behaviour into consideration.

There are several ways of dealing with this issue. Nierstrasz [1995] uses the traces of menus offered by objects as a notion of behavioural types and proposes a notion of subtyping, *request substitutability*, that generalizes the Liskov Substitution Principle [Liskov and Wing 1994]. This substitution principle requires that whenever S is a subtype of T, we can replace an object of type T with another object of type S; the resulting program will still have the behaviour of the original program. This means that a service can be refined as long as the original promises are still kept. According to the extension relation defined by Brinksma et al. [1995], request substitutability gives rise to a pre-order which is close to the failures model of CSP [Hoare 1985].

Colaço et al. [1997; 1999] define an actor calculus supporting non-uniform objects. The process language is inspired both by the (functional) Object Calculus [Abadi and Cardelli 1996] and by Typed Concurrent Objects (an asynchronous π-calculus with input-guarded labelled sums and output selections [Vasconcelos 1994]), following the actor model [Hewitt et al. 1973]. The authors define a type system that detects "orphan messages". These are messages that may fail to ever be accepted by any actor, because dynamic changes to the interface of the actor cause the service requested not to be available anymore. Types describe interfaces annotated with multiplicities (that is, how often a method can be invoked), and the type system involves complex operations on a lattice of types. The authors give a type inference algorithm for this type system, based on the familiar notion of solving set constraints by resolution.

Najm and Nimour [1997] define another calculus of non-uniform objects, based on the asynchronous π-calculus. The calculus is complemented by a type system [Najm and Nimour 1997; Najm et al. 1999a; Najm et al. 1999b] that handles dynamic method offers in interfaces and guarantees a liveness property, namely that every pending request will eventually be processed. In this case, a type is defined as a set of deterministic guarded parametric equations that describe a transition system (which may be infinite). The type system has notions of type equivalence, compatibility and subtyping defined using this transition system approach; in particular the notion of subtyping is based on the notions of strong simulation and strong bisimulation and is decidable. Moreover, the authors are able to define a type inference algorithm for their system.

Puntigam [2001a; 2001b] and Puntigam and Peter [2001] also define a calculus of non-uniform objects in which process types impose constraints on the ordering of messages. A static type inference system ensures that even when the set of acceptable messages changes dynamically for an object, every message sequence sent to it will eventually be received.

Several authors adopted the processes-as-types approach (see Section 2.5) to deal with the issue of non-uniform objects. Ravara and Vasconcelos [2000] developed a behavioural type system for TyCO [Vasconcelos 1994] to ensure the absence of 'message-never-understood' errors in non-uniform concurrent objects (the property is an adaptation of the usual 'message-not-understood', as a message can momentarily be not understood due to the non-uniform method availability). The type safety result guarantees that every message has a chance of being received if it requires a method that may become enabled at some point in the future. The type language is the process algebra ABT [Ravara et al. 2012].

More recently, Caires and Seco [2013] introduce the concept of *behavioural separation*. Behavioural separation is a general principle for controlling interference in

concurrent, higher-order imperative programs (written in languages such as ML or Java). Behavioural separation types combine notions originating in behavioural type theories, separation logics, and behavioral-spatial types [Caires 2008]. They make it possible to enforce fine-grained interference control disciplines and at the same time preserve compositionality, information hiding, and flexibility. Behavioural separation types specify how the values of a program can be used safely by client code, by integrating behavioural operations such as parallel ($T \mid U$) and sequential ($T\,;U$) composition and intersection ($T \,\&\, U$) within a substructural type theory. Basic functional ($T \mapsto V$) and qualification ($l{:}T$) types describe single usages of a value as a function and as a record, respectively. For example, the safe usage for a dictionary abstract data type d for key type K and value type V can be specified by the behavioural separation type assertion

$$d : \mathtt{rec}(X)(\mathit{assoc} : K \mapsto V \mapsto \mathsf{0} \,\&\, !(\mathit{find} : K \mapsto V))\,;X$$

This specifies that at each moment either a single client of d can call the assoc operation, or an arbitrary number of clients (! denotes shared resources) can concurrently call the find operation. On the other hand, a lock-serialised dictionary c will be well-typed under the more flexible type

$$c : !(\mathit{assoc} : K \mapsto V \mapsto \mathsf{0}) \mid !(\mathit{find} : K \mapsto V)$$

that allows an arbitrary number of clients to concurrently and safely call either operation.

2.5. Processes as types

Another approach, originating from work on type and effect systems, is that of considering processes as types. Here, types are processes that are sound abstractions of the behaviour of programs, and an analysis of the type thus becomes an analysis of the behaviour of the program. Since program properties are checked at the level of types, not programs, these properties are often decidable, and therefore this approach can benefit from the advantages of type checking as well as model checking. Similarly, session types can be seen as processes abstracting the protocol followed by programs.

Boudol [1997] describes a dynamic type system for the Blue Calculus, which is a version of the π-calculus that directly incorporates the λ-calculus. In this type system, types are functional types in the sense of the simply typed λ-calculus but now also incorporate a version of recursive Hennessy-Milner logic in which modalities are interpreted as named resources. In this type system, types are inhabited by processes, and the type system is able to express a form of causality in the way names are used within a process. This ensures that messages sent to a name will meet a corresponding offer.

Kobayashi [2000] and Kobayashi et al. [2000] study type systems for detecting deadlock and livelock in a synchronous π-calculus. In these systems, the type of a channel carries information about both the arity of the channel and its *usage*. The usage contains information about the admissible sequences of input and output actions, about when the channel may be used, and whether it must be used.

Igarashi and Kobayashi [2004] describe a so-called Generic Type System. This is a general framework that makes it possible to develop type systems that capture various properties of π-calculus processes. In the Generic Type System, types are processes from the restriction-free fragment of CCS and the type system involves a general subtype relation and a consistency condition on types. Type systems for concrete properties can now be obtained from the generic type system by instantiating the general subtyping relation and the consistency condition. The properties of the particular type system follow from the general properties of the Generic Type System. All one needs

to prove is that reductions on types preserve consistency and that consistency on types implies the desired condition on processes. The Generic Type System is able to capture specific type systems for safety properties (including arity mismatch, race freedom and even deadlock freedom), but not liveness properties.

2.6. Interface automata

Interface automata [de Alfaro and Henzinger 2001] constitute an automata-based approach to behavioural types for specifying extensional program properties. Interface automata are now used to specify interfaces of components, and a *refinement* relation is used to compare abstract and concrete interface specifications. In much of this work the focus is not on establishing explicit typing rules for an underlying programming language but instead on defining notions of conformance, compatibility and composition. Interface automata are similar in aim to session types and behavioural contracts, but their automata-based presentation makes the two approaches technically different.

Lee and Xiong [2004] give a behavioural type system based on interface automata for the Ptolemy II framework (for composing concurrent components) that captures the dynamic interaction in an environment for component-based design. The interaction types and component behaviour are given as interface automata, and type checking is carried out via composition of automata. Chouali et al. [2010] present a formal approach, based on interface automata and protocol specifications, that allows one to adapt components and eliminate possible behavioural mismatches that occur in interactions. The approach ensures that components can be re-used in diverse situations without their code being affected. Chouali and Hammad [2011] describe an approach that uses a combination of component models and interface automata to assemble components and to formally verify that they are interoperable.

Carrez et al. [2003] define the notions of behavioural interfaces, contracts and contract satisfaction as basis for studying the sound assembly of components.

3. BINARY SESSIONS

This section describes session types and contracts for communications involving exactly two participants. We first discuss input/output types and linear types, which have been a main source of inspiration for this line of work.

3.1. Input/output types and linear types for the π-calculus

The simplest type system for the π-calculus, [Milner 1993], only keeps track of the number of arguments channels may carry, thus preventing "arity mismatches" where the number of channels sent by a client differs from the number expected by a server. Types are tuples (of types again) of the form $(T_1, \ldots, T_n)$ describing communication channels able of carrying n (with $n \geq 0$) channels of types T_1 to T_n. For example, a type $(\mathsf{nat}, (\mathsf{nat}))$ describes a channel on which a server expects a pair of values, composed of a natural number and of a channel on which it may reply another natural number (e.g., its successor).

A type environment Γ is a map (a partial function of finite domain) from names to types, and one can think of a type judgement $\Gamma \vdash P$ as stating that the behaviour of process P is given by the type information in Γ.

This kind of type system can be refined by including more information on how channels are used in computations. One such refinement, input/output types [Pierce and Sangiorgi 1996], includes an optional *polarity* (or directionality) in each type, so that a type of the form $?(T_1, \ldots, T_n)$ can only be used for input (processes can only read on the associated channel) and a type $!(T_1, \ldots, T_n)$ can only be used for output (processes can only write on the channel). The type of the server above can now be refined as

$?(\mathsf{nat}, !(\mathsf{nat}))$, so that the channel can only be used to read a pair of values, the second of which can only be used to write a natural number.

Such a refinement is useful not only to prevent programming mistakes (where for example the server after reading the pair of channels, tries again to read on the second channel, thus leaving the client forever waiting for the reply), but also provides for more powerful reasoning techniques. The input/output system was used to prove the preservation of beta-equivalence on a translation [Milner 1992] of the λ-calculus into the π-calculus.

A further refinement introduces *multiplicities* on top of polarities, yielding what is known as the linear π-calculus [Kobayashi et al. 1999]. The type system uses ideas from linear logic [Girard 1987] on controlling the number of times a hypothesis can be used in a proof. In the linear π-calculus type system, the multiplicity of a channel controls the number of times it can be used. For example, a channel of type $!^1(T_1, \ldots, T_n)$ can only be used once (and for output) and is called *linear*. A type $?^\omega(T_1, \ldots, T_n)$ can be used zero or more times (for input only). We can then define a notion of type addition $T_1 + T_2$; this operation is partial and only defined for types of the same multiplicity and for appropriate polarities. For instance, we have that $!^1(T_1, \ldots, T_n) + ?^1(T_1, \ldots, T_n) = \{?, !\}^1(T_1, \ldots, T_n)$.

We are now in a position to refine the type of our server. A type $?^\omega(\mathsf{nat}, !^1(\mathsf{nat}))$ describes a channel that can be read multiple times: the server is supposed to serve multiple requests on a same channel. The second value in one such message is a channel that must be used exactly once for output: servers are expected to reply exactly once to each request. Communication on linear channels cannot be interfered by other computations, and cannot affect these. This provides for further process equivalences that could not be established with weaker type systems.

In the linear type discipline, we define addition of type environments by a pointwise extension of type addition.

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) + \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \setminus dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \setminus dom(\Gamma_1) \end{cases}$$

If we read the above from the right to left, this describes how a type environment can be split into two sub-environments. The rule for typing a parallel composition $P_1 \mid P_2$ is crucial. We must split the resources in Γ into the resources Γ_1 used to type P_1 and the resources Γ_2 used to type P_2.

$$\frac{\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \quad \text{where } \Gamma = \Gamma_1 + \Gamma_2$$

This type rule is directly inspired by the proof rule for multiplicative conjunction in linear logic and versions of it also appear in the binary session type systems that we shall now describe.

3.2. Binary session types

Session types further refine the linear type system introduced in the previous section. Here we are concerned with *binary* session types, types that describe communication patterns involving exactly two participants. A binary session type describes a protocol as seen from the point of view of one of the two participants.

The basic constructs denote the two contributions to a message exchange. One writes $!\mathsf{nat}$ to denote the *output* of a natural number, and one writes $?\mathsf{nat}$ to denote the *input* of a natural number. Types may be composed by means of a *prefix* operator. If T is a type, then $!\mathsf{nat}.T$ is also a type, and denotes an interaction that starts with the output of a

natural number, followed by the behaviour prescribed by T. The completed protocol, that is, the protocol on which no further interaction is possible, is denoted by end. Putting all these pieces together one can write a session type

$$!\mathsf{nat}.?\mathsf{bool}.\mathsf{end}$$

describing a series of message exchanges, starting with the output of a natural number, followed by the input of a boolean value, followed by termination of the protocol.

The above type describes an interaction as seen from one of the participants' point of view. The type for the second participant is the complementary, or the *dual*, obtained as follows. The dual of output is input, the dual of input is output, and the dual of end is end. In either case, input or output, the types of the values exchanged in messages remain unchanged. In this way, the dual of $!\mathsf{nat}.?\mathsf{bool}.\mathsf{end}$ is $?\mathsf{nat}.!\mathsf{bool}.\mathsf{end}$.

A further important construct usually present in session types is *choice*. Again we have two points of view: that of a participant that offers the menu of options, and that of the participant that selects a particular option. A choice between depositing or withdrawing money at some ATM can be written, from the point of view of the client as $\oplus\{\mathit{deposit}\colon T_1, \mathit{withdraw}\colon T_2\}$, where T_1 describes the behaviour subsequent to the selection of the deposit operation, and T_2 that after the selection of withdraw. The ATM, on the other hand, offers a menu, written as $\&\{\mathit{deposit}\colon T_3, \mathit{withdraw}\colon T_4\}$. Here, T_3 and T_4 describe the behaviours after the reception of a deposit or a withdraw operation. Duality also applies to choice. The above two types are dual if T_1 is a dual of T_3, and T_2 is a dual of T_4.

So far session types can only offer series of fixed-length interactions. Often the exact number of messages exchanged (and choices performed) cannot be determined in advance. Just think of the type describing a typical session between a client and an ATM: after selecting option withdraw and providing the amount, the client would like to have all her choices available again, so that, e.g., she may then check the remaining balance. Potential infinite behaviour in session types is usually introduced by means of recursion operator: if T is a type and a is a type variable, then $\mathsf{rec}\, a.T$ and a are also types. The duality relation for recursive types is defined co-inductively [Gay and Hole 2005]. We are now in a position to introduce the type T_{ATM} for our ATM machine, as seen from the point of view of a client. Clients start by providing their user-id in the form of a string; the authentication details are omitted. They are then offered a four-way menu. If $\mathit{withdraw}$ is picked, then clients must provide a nat describing the amount to be withdrawn, to which the ATM then answers with a $\mathit{dispense}$ or $\mathit{overdraft}$ option. In either case, the client is again provided with the four-choice menu.

$$\begin{aligned}!\mathsf{string}.\mathsf{rec}\, a.\, \oplus\{&\mathit{deposit}\colon !\mathsf{nat}.a\\ &\mathit{withdraw}\colon !\mathsf{nat}.\&\{\mathit{dispense}\colon a, \mathit{overdraft}\colon a\},\\ &\mathit{balance}\colon ?\mathsf{nat}.a,\\ &\mathit{quit}\colon \mathsf{end}\}\end{aligned}$$

From the point of view of the ATM machine the type is the dual, which can be obtained by swapping $\oplus$ with $\&$ and ! with ? in the type above.

The type for the ATM, even if revealing, is first order: the messages exchanged by the client and the ATM server are composed of uninterpreted labels and natural numbers. A more interesting type theory, introduced by Honda et al. [1998], allows for higher-order messages, that is, for messages to carry arbitrary session types. The phenomenon is called *delegation*, and remains today the norm in session type systems. Suppose the authentication against the ATM server and the subsequent interaction are to be performed by different processes. We can arrange our computation so that an initial process conducts the authentication part of the protocol, and then *delegates*

the communication channel to the second process. The channel on which delegation is performed may be of type $!(\mathsf{rec}\, a.\, \oplus \{\mathit{deposit}\colon !\mathsf{nat}.a, \ldots, \mathit{quit}\colon \mathsf{end}\}).\mathsf{end}$. The interplay between delegation and recursion in the definition of duality is discussed by Bernardi and Hennessy [2014].

The types discussed so far describe sessions, that is interactions meant to be run without interference. These types are usually called *linear*. Complementary to these, we need types whose objects may be *shared* and that can in particular be used to establish new (linear) sessions. Note that shared types may be communicated over linear types, but the continuation of a linear type is, in most of the approaches, a linear type. When it comes to shared types there are a few variants in the literature. For Gay and Hole [2005], a shared type S is either a base type, such as nat or bool, or else a type $\hat{}[T]$ describing an object capable of carrying a session of type T. As an example, the shared name of an ATM is of type $\hat{}[T_{\text{ATM}}]$. Honda et al. [1998] make it explicit that the type is capable of generating both session ends, by writing instead $\langle T_{\text{ATM}}, T'_{\text{ATM}}\rangle$, where T'_{ATM} is a dual of T_{ATM}.

Vasconcelos [2012] eliminates the stratification of types into linear and shared, by classifying each prefix with a lin (linear) or un (unrestricted or shared) qualifier. The type of the shared name of the ATM becomes $\mathsf{rec}\, b.\mathsf{un}!T_{\text{ATM}}.b$. Eliminating stratification allows for describing channels that start as linear and end as unrestricted (see examples by Vasconcelos [2012]). In order to capture within a single type the capabilities of both ends of a channel, Giunti and Vasconcelos [2016] use *pair types* (T_1, T_2) where T_1 describes the behaviour of one end and T_2 the behaviour of the other. Contrary to Honda et al. [1998], types T_1 and T_2 need not be dual of each other. Brogi et al. [2004] and Vallecillo et al. [2006] use binary session types to study the safe interaction of software components.

3.3. π-calculi for binary session types

Types need programming languages. Session types were initially developed in the realm of the π-calculus. They have since then been incorporated in functional and object-oriented languages (early references are Dezani-Ciancaglini et al. [2005] and Vasconcelos et al. [2006]). Here we focus on the π-calculus.

Session types require mild variations of the π-calculus as introduced by Milner et al. [1992]. In the π-calculus, types are assigned to channels. If channel x is of type $!T_1.T_2$ and value v is of type T_1, then one may write $x!v.P$ to denote a process that writes v on x and continues as prescribed by process P. The type of x in P is T_2. Conversely, if x is of type $?T_1.T_2$, then $x?y.P$ denotes a process that reads a value from channel x, binds it to y, and continues as P. In P, channel y is of type T_1 and x is of type T_2.

Processes are typed against *typing contexts*, essentially a map from channels to types. *Sequents* are of the form $\Gamma \vdash P$ and say that process P is well typed under context Γ. Following the description above, the typing rules for input and output (of linear values on linear channels) [Honda et al. 1998] should be easy to understand.

$$\frac{\Gamma, x\colon T_2 \vdash P}{\Gamma, y\colon T_1, x\colon !T_1.T_2 \vdash x!y.P} \qquad \frac{\Gamma, y\colon T_1, x\colon T_2 \vdash P}{\Gamma, x\colon ?T_1.T_2 \vdash x?y.P}$$

In order to deal with choice, Honda et al. [1998] introduce two new language constructs, called *branch* and *select*. If x is of type $\&\{l_1\colon T_1, l_2\colon T_2\}$, then $x \rhd \{l_1\colon P_1, l_2\colon P_2\}$ denotes a process (usually called *branching*) that offers two options, and behaves as P_1 if option l_1 is selected and as P_2 if option l_2 is selected. Conversely, if x is of type $\oplus\{l_1\colon T_1, l_2\colon T_2\}$, then $x \lhd l_2.P$ *selects* the l_2 option on channel x and proceeds as P.

The fundamental change to the π-calculus often required by session types forces the syntactic distinction of the two ends of a channel. Gay and Hole [2005] write x^+ and x^- to speak about the two ends of channel x. In the π-calculus [Milner et al. 1992] the

parallel composition of processes P_1 and P_2 is denoted $P_1 \mid P_2$, and channels are created by means of the ν constructor, as in $(\nu x\colon T)P$. Gay and Hole [2005] write $(\nu x\colon T)P$ and use x^+ of type T and x^- of a type dual of T in process P, as in

$$(\nu x\colon \mathsf{!nat}.U)(x^+!5.P_1 \mid x^-?z.P_2)$$

which reduces in one step to $(\nu x\colon U)(P_1 \mid P_2[5/z])$, where $P_2[5/z]$ denotes process P_2 with the free occurrences of channel z replaced by the value 5. An alternative formulation uses (non-annotated) identifiers to describe the two ends of a channel [Vasconcelos 2012]. In this case, when creating a new channel we explicitly name its two ends using a pair of conventional variables, as in $(\nu xy\colon \mathsf{!nat}.U)(x!5.P_1 \mid y?z.P_2)$.

The syntactic distinction between the two ends of channels is required only when both the situations below arise together:

(1) Processes may obtain both ends of a channel and use them in *sequence*, as in $x?y.x!y.y?z$, where the first x denotes one end whereas y denotes the other end of a same channel, and
(2) Types describe only one end of a channel, as in $\mathsf{!nat.end}$.

Yoshida and Vasconcelos [2007] and Giunti and Vasconcelos [2016] further discuss the problem. The type system by Honda et al. [1998] does not require the distinction between the two channel ends because the particular nature of channel passing (bound output) precludes processes from obtaining both ends of a channel, thus avoiding condition (1). The same happens in the interpretation of session types in intuitionistic linear logic discussed in Section 6.1. The system by Giunti and Vasconcelos [2016] uses types that describe *both* ends of a channel, as in $(\mathsf{!nat.end}, \mathsf{?nat.end})$, thus steering clear from condition (2).

The type system just sketched describes the behaviour of the (free) channels in programs, thus providing an abstract description of the communication patterns of a program. In addition it prevents certain kinds of *run-time errors*, described in Section 7.1.

3.4. Binary contracts

Contracts take an approach different from session types, by using process algebra-like languages or labelled transition systems for describing abstractions of the communication behaviour of programs. Before reviewing the most relevant literature, we present a simple example of a contract-like description of the binary interaction between a client and an ATM in the example reported in Section 3.2. The behaviour of the ATM from the point of view of the server is described as follows.

$$?\mathit{auth};(?\mathit{deposit};?\mathit{amount}+?\mathit{withdraw};?\mathit{amount};(!\mathit{dispense}+!\mathit{overdraft}) + ?\mathit{balance};!\mathit{amount})^*;?\mathit{quit}$$

While the prefixes in a binary session type describe the types that are communicated or (in the case of branching and selection) the choices that are possible, the labels in a contract directly describe the actions allowed by the ATM.

The input action auth identifies the initial authentication data sent by the client to the ATM. After this step, the ATM enters a cycle offering three functionalities: $\mathit{deposit}$ an amount indicated by the client, $\mathit{withdraw}$ or show the current $\mathit{balance}$. The cycle is terminated by the quit action. The behaviour of a client interested in asking for the balance and then performing a withdraw action can be represented as follows.

$$!\mathit{auth};!\mathit{balance};?\mathit{amount};!\mathit{withdraw};!\mathit{amount};(?\mathit{dispense}+?\mathit{overdraft});!\mathit{quit}$$

Intuitively, the two contracts are compatible in the sense that their combination guarantees the completion of the expressed protocols. The papers on contracts that we review below formalize appropriate notions of compatibility between contracts.

Fournet et al. [2004] introduce the notion of contract as the description of the input-output behaviour of processes. They use the process calculus CCS to denote contracts; a main contribution of this work is the formalization of the notion of *stuck-free conformance*: a CCS process P conforms to a contract C if P can replace C in every context preserving stuck-freedom (roughly, stuck-freedom corresponds to absence of local deadlocks). Conformance checking is not decidable for the full CCS calculus but the authors show how conformance checking can in some cases be handled using a model checker.

This approach has inspired several subsequent works. Carpineti et al. [2006] consider a similar language for the description of contracts and processes. They introduce a different notion of conformance, namely, an asymmetric client-service compliance notion: a client and a service are compliant when in every computation the client is guaranteed to reach a successful state. Such a kind of contracts has been subsequently extended in two directions: Laneve and Padovani [2007] introduce the notion of input and output alphabet associated to a contract, while Castagna et al. [2009b] propose dynamic filters that can be associated to services in order to eliminate interactions on non admitted channels. Both approaches aim at relaxing the conformance relation, thus extending the set of processes that can safely replace a given service. Variants of this approach deal with dynamic communication topologies [Castagna and Padovani 2009] or with the standard languages for the description and composition of Web Services [Laneve and Padovani 2013].

3.5. Variations and extensions of binary sessions

Different session calculi have been proposed in the literature, either by extending the types above with new features or by changing the operational semantics of the underlying languages.

Asynchronous semantics. Gay and Vasconcelos [2010] study a functional language with asynchronous (buffered) semantics. In the context of the π-calculus with sessions, Kouzapas et al. [2016] consider a semantics based on order-preserving asynchronous communication inside each session and asynchronous message arrival for general channels.

Event-driven programming. Kouzapas et al. [2016] extend session types with non-blocking detection of message arrival (events) and dynamic inspection of session types to model event-driven programming. As a result, they can encode event selectors, a central component of event-driven systems, enabling the development of type-safe event-driven applications. They also define a systematic transformation from multithreaded to event-driven processes which is type- and semantics-preserving.

Exceptions. Carbone et al. [2008] extend binary sessions with a throw primitive to raise exceptions, and exception handlers for handling them. Exceptions that require coordinate handling from both the session participants are considered. Both safety and liveness properties are ensured.

Service-oriented programming. Service-Oriented Computing (SOC) applications are generated by dynamically looking for available services on the network, and composing them so as to obtain the desired functionalities. The different services communicate by exchanging messages over the network. While current standards only check syntactic compatibility of service signatures, session types have been proposed to ensure also behavioural compatibility. Some calculi to model SOC systems have been proposed.

— SCC [Boreale et al. 2006] was the first attempt to define a calculus able to directly model SOC systems in a π-calculus style. In particular, SCC features service definition and invocation as primitive operators. When a service is invoked, a private

session is created to allow communication between the two processes. Results computed inside the session are propagated to the upper level using a return primitive. This primitive alone is however not enough to model the complex patterns needed to coordinate different client/service pairs. New calculi have been developed to address this issue. A type system based on session types to guarantee deadlock freedom in SCC is presented by Bruni and Mezzina [2008].

— CASPIS [Boreale et al. 2008] extends SCC with pipelines, allowing the definition of flows of data between services, thus improving the modeling of complex communication patterns.

— SSCC [Cruz-Filipe et al. 2014; Lanese et al. 2007] also extends SCC aiming at easier modelling of SOC patterns. It uses streams (rather than pipelines), a concept orthogonal to the session hierarchy. Session types ensuring correctness of session communication are presented by Lanese et al. [2007]. A simpler type system, ensuring sequentiality of communication, is exploited by Cruz-Filipe et al. [2008] to enable program transformations to break large sessions into smaller ones.

Other approaches to modelling services use multiparty sessions and are presented in the next section.

4. MULTIPARTY SESSIONS

Binary session types, described in the previous section, are restricted to communication patterns involving exactly two participants. Multiparty session types drop this restriction. In many cases, it is possible to describe and reason about multiparty conversation patterns by means of a composition of binary sessions. However, there are also patterns involving more than two communicating parties for which binary sessions do not suffice, and multiparty session types are needed.

4.1. Global and local types

Consider by way of example an extension of the ATM example from Section 3, where the ATM establishes a different session with the bank central server reporting any operation that the client chooses. Intuitively, the ATM needs to contact the bank only after the client has made a choice, and never before. Unfortunately, since sessions are binary such a constraint cannot be imposed at type level. For example, a well-typed implementation of the ATM could be a process that, independently from which branch is selected, always reports to the bank that the client has made a deposit.

To address this problem, Honda et al. [2008] propose a generalisation of binary session types called multiparty session types. Multiparty session types provide for *global descriptions* of interactive behaviour. Under this paradigm, a software architect prepares a *global view* of all the message exchanges that take place, instead of separately defining the behaviour of each individual channel endpoint (as in binary session types, where we only specify the behaviour of each side of a binary session). The local behaviour of each endpoint can be mechanically obtained from the global description by applying a *projection* operation. A global description is therefore a "formal blueprint" of how a communicating system should behave and it provides a concise specification of how messages flow within the system. This should have a major impact on software quality, since a global description will:

(1) decrease the risk of introducing programming errors;
(2) make it easier to detect such errors (both manually and by automatic means), and
(3) guarantee the absence of deadlocks.

One can use global descriptions at different levels of abstraction, ranging from abstract descriptions of *protocols* (multiparty session types, described below) to descriptions

of concrete implementations (as done, e.g., by Carbone and Montesi [2013]). As an example, the following is a global type describing a session with three participants—Client, ATM and Bank—where the ATM correctly reports to the bank all the choices made by the client:

$$\mathsf{Client} \to \mathsf{ATM}(\mathsf{string}).\ \mathsf{rec}\, a.\ \mathsf{Client} \to \mathsf{ATM} \left\{ \begin{array}{l} \mathit{deposit}\colon \mathsf{Client} \to \mathsf{ATM}(\mathsf{nat}).\ \mathsf{ATM} \to \mathsf{Bank}\{\mathit{deposit}\colon a\}, \\ \mathit{withdraw}\colon \mathsf{Client} \to \mathsf{ATM}(\mathsf{nat}). \\ \quad \mathsf{ATM} \to \mathsf{Bank} \left\{ \mathit{withdraw}\colon \mathsf{ATM} \to \mathsf{Client} \left\{ \begin{array}{l} \mathit{dispense}\colon a, \\ \mathit{overdraft}\colon a \end{array} \right\} \right\}, \\ \mathit{balance}\colon \mathsf{ATM} \to \mathsf{Client}(\mathsf{nat}).\ \mathsf{ATM} \to \mathsf{Bank}\{\mathit{balance}\colon a\}, \\ \mathit{quit}\colon \mathsf{ATM} \to \mathsf{Bank}\{\mathit{quit}\colon \mathsf{end}\}. \end{array} \right\}$$

The multiparty session type above (or simply *global type*) specifies in which order the implementation of the client, the ATM and the bank have to exchange messages and the order of requests that are involved. The key operations in a global type are interactions such as

$$\mathsf{Client} \to \mathsf{ATM}(\mathsf{string})$$

in which a sender (Client) sends a message of some type (string) to a receiver (ATM), and choices such as

$$\mathsf{ATM} \to \mathsf{Client}\{\mathit{dispense}\colon \ldots, \mathit{overdraft}\colon \ldots\}$$

in which a sender (ATM) asks a receiver (Client) to select a certain branch.

Global types are used for checking programs running in parallel, each implementing one of the roles specified in the type, e.g., Client, ATM, and Bank. In order to realise that, a notion of projection from global types to local types is defined. For example, the local type corresponding to ATM in the interaction above would be:

$$\mathsf{Client}\,?\mathsf{string}.\ \mathsf{rec}\, a.\mathsf{Client}\,\& \left\{ \begin{array}{l} \mathit{deposit}\colon \mathsf{Client}\,?\mathsf{nat}.\ \mathsf{Bank} \oplus \{\mathit{deposit}\colon a\}, \\ \mathit{withdraw}\colon \mathsf{Client}\,?\mathsf{nat}. \\ \quad \mathsf{Bank} \oplus \left\{ \mathit{withdraw}\colon \mathsf{Client} \oplus \left\{ \begin{array}{l} \mathit{dispense}\colon a, \\ \mathit{overdraft}\colon a \end{array} \right\} \right\} \\ \mathit{balance}\colon \mathsf{Client}\,!\mathsf{nat}.\ \mathsf{Bank} \oplus \{\mathit{balance}\colon a\}, \\ \mathit{quit}\colon \mathsf{Bank} \oplus \{\mathit{quit}\colon \mathsf{end}\}. \end{array} \right\}$$

Note how local types are slightly more complex than standard binary session types since each communication operation is now labeled with the party this role is supposed to communicate with (rather than ?nat we now write Client?nat).

An important question when dealing with multiparty sessions is that of finding a suitable language for describing these interactions. Honda et al. [2008] present a generalisation of binary sessions to multiparty asynchronous sessions for the π-calculus. In contrast to the binary case, sessions are now established between multiple processes via multiparty synchronisation. Then, private (in-session) communication is carried out, asynchronously, between session participants. Technically, sessions are established as follows:

$$a_{2..n}?\tilde{s}.P_1 \mid a_2!\tilde{s}.P_2 \mid \cdots \mid a_n!\tilde{s}.P_n \ \to\ (\nu\tilde{s})(P_1 \mid \cdots \mid P_n \mid\ s_1\colon \emptyset \mid \cdots \mid\ s_m\colon \emptyset)$$

In the above, the term on the left-hand side of the reduction $\to$ contains n processes running in parallel, each of them willing to establish a session on public channel a. Note how each participant is labelled with a role name $(1, 2, \ldots, n)$. Each label (where 1 is actually denoted by $2..n$, to clarify the number of expected processes) corresponds to the unique role that a participant in the new session has to play. The session is

established through a distributed synchronisation which creates the session (private) channels $s_1, \ldots, s_m$ and the corresponding FIFO queues (denoted in the reductum by $s_1 : \emptyset, \ldots, s_m : \emptyset$). Once the connection is established, processes $P_1, \ldots, P_n$ can asynchronously communicate by using the queues corresponding to one of the channels $s_1, \ldots, s_m$.

Coppo et al. [2016] simplify the approach of Honda et al. [2008], by making the number of session channels created upon session initiation no longer arbitrary, but dependent on the number of session roles. In particular, session initiation creates a session channel for each ordered pair of roles.

4.2. Conversation types

Conversation types [Caires and Vieira 2010] extend (binary) session types to deal with multiparty interaction. Although conversation types were originally introduced to type terms in the Conversation Calculus [Vieira et al. 2008], the approach carries over to a more foundational setting, namely to a modest extension of the π-calculus in which communication actions are *labelled*. Given that a session type characterises the usage of a single channel by two parties, it seems natural to consider that a multiparty extension of a session type characterises the usage of a single channel by multiple parties. To motivate the underlying model, consider the following example where three (concurrent) threads interact in a channel $bank$: the leftmost thread sends 100, the middle thread sends true, and the rightmost thread sequentially receives two values.

$$bank!100 \mid bank!\mathsf{true} \mid bank?x.bank?y$$

Looking at the specification one may immediately identify a potential communication problem: two threads are simultaneously trying to send a message, a communication *race*. As a consequence, the receiving process may actually receive first either value 100 or value true, making it impossible to (statically) characterise how the received values can be used. Now consider that we extend the specification above, by adding labels to communication actions.

$$bank!deposit(100) \mid bank!letOverdraft(\mathsf{true}) \mid bank?deposit(x).bank?letOverdraft(y) \quad (1)$$

Thanks to the labels we may now distinguish two synchronisations, given the order imposed by the receiving process: first a $deposit$ labelled message is exchanged, then a $letOverdraft$ labelled message. Labels thus allow one to recover pairwise linear interactions, even if multiple parties share a single communication medium. The use of labels allow one to avoid communication errors and race errors.

Let us now turn to a typing characterisation of the system given in (1). The leftmost process uses channel $bank$ to output an integer, which we may characterise with the (session) type $!\mathsf{Int}.\mathsf{end}$. Extending the type with the corresponding label we then have the conversation type $!deposit(\mathsf{Int}).\mathsf{end}$, and likewise for the process in the middle we have $!letOverdraft(\mathsf{Bool}).\mathsf{end}$. On the receiving end, the rightmost process may be characterised by type $?deposit(\mathsf{Int}).?letOverdraft(\mathsf{Bool}).\mathsf{end}$. The sequential exchange of messages $deposit$ and $letOverdraft$ in channel $bank$ is captured by conversation type $\tau deposit(\mathsf{Int}).\tau letOverdraft(\mathsf{Bool}).\mathsf{end}$, where each τ captures a message exchange internal to the characterised system.

We may draw a comparison between the conversation types described above and the local and global types of Honda et al. [2008], described in Section 4.1. The conversation types include at the same level both the type of interactions internal to the system (via τ), specified in global types by Honda et al. [2008], and interactions between the system and the external environment (via output $!$ and input $?$), corresponding to local types in [Honda et al. 2008]. For the sake of illustration consider the system below, consisting

of part of (1).

$$bank!letOverdraft(\mathsf{true}) \mid bank?deposit(x).bank?letOverdraft(y)$$

Channel $bank$ is used according to type $?deposit(\mathsf{Int}).\tau letOverdraft(\mathsf{Bool}).\mathsf{end}$, saying that first a $deposit$ message is received after which message $letOverdraft$ is exchanged. Such type is obtained as a behavioural combination of two (local) types, namely $!letOverdraft(\mathsf{Bool}).\mathsf{end}$ and $?deposit(\mathsf{Int}).?letOverdraft(\mathsf{Bool}).\mathsf{end}$. Notice that the τ combines the (dual) output and input descriptions for message $deposit$, while the reception of message $letOverdraft$ is left at the interface level, open to synchronise with an output originating from the external environment.

This ability to *merge* behaviours (or symmetrically, the ability to *split* behaviours into smaller pieces), defined in an algebraic way by Caires and Vieira [2010], allows one to compositionally characterise systems. In fact, the behaviour of a system can be obtained by merging the behaviours of its components. Furthermore, the ability of splitting behaviours allows one to address configurations in which parties engage dynamically in conversations: a participant may decide to split its behaviour in two parts, execute one of them, and delegate the second one to a participant which joins the conversation dynamically. This possibility is the key to model multiparty interaction. Indeed, since the underlying model (the labelled π-calculus) does not support atomic multiparty synchronisation, the way in which multiparty interaction is modelled is by allowing multiple parties to join an ongoing conversation (realised by channel name passing). Using the terminology from session types, conversation joining is supported by channel *delegation*, only now delegation is partial: the delegating party will still have access to the communicated channel.

Baltazar et al. [2012a] introduce a novel type construct to capture the idea that some behaviours are not necessarily carried out immediately and can actually take place *sometime* in the future. Type $\Diamond!letOverdraft(\mathsf{Bool})$ says that the output of message $letOverdraft$ will happen sometime but not necessarily immediately. The $\Diamond$ type constructor is related to the “eventually" operator from temporal logic and satisfies expected laws such as $B <: \Diamond B$ (see Baltazar et al. [2012a]).

When composing the (sometime) output of message $letOverdraft$ and the (immediate) output of message $deposit$ we obtain the type $!deposit(\mathsf{Int}).\Diamond!letOverdraft(\mathsf{Bool}).\mathsf{end}$ that composed with the dual (sequential) capabilities $?deposit(\mathsf{Int}).?letOverdraft(\mathsf{Bool}).\mathsf{end}$ yields the global protocol

$$\tau deposit(\mathsf{Int}).\tau letOverdraft(\mathsf{Bool}).\mathsf{end}$$

4.3. Multiparty contracts

Contracts for multiparty process composition were initially investigated by Bravetti and Zavattaro [2007] using a *choreography language* to describe the globally observable behaviour of correctly interacting peers. A choreography language, like WS-CDL [Kavantzas et al. 2005] or its formalization [Busi et al. 2005], is a language having an interaction, namely a communication between two participants, as the main building block. Below we use choreography languages to write contracts, that is abstract descriptions of program behaviour. A contract written in a choreography language is called a *choreography*. Choreography languages can also be used to write global types, as seen in Section 4.1, or as a programming language, as done, e.g., by Carbone and Montesi [2013]. As a simple example of choreography, consider the binary interaction between the client (denoted by C) and the ATM (denoted by A) described in Section 3.4 expressed in the choreography language proposed by Bravetti and Zavattaro [2007]. In that example, we showed an ATM contract allowing potential clients to repeatedly perform $deposit$, $withdraw$ or $balance$ operations, and a client contract simply performing a

balance followed by a *withdraw* request. The combination of these two local behaviours generates the following choreography.

$$auth_{\mathsf{C}\to\mathsf{A}}; balance_{\mathsf{C}\to\mathsf{A}}; amount_{\mathsf{A}\to\mathsf{C}}; \\ withdraw_{\mathsf{C}\to\mathsf{A}}; amount_{\mathsf{C}\to\mathsf{A}}; (dispense_{\mathsf{A}\to\mathsf{C}} + overdraft_{\mathsf{A}\to\mathsf{C}}); quit_{\mathsf{C}\to\mathsf{A}}$$

The notation $auth_{\mathsf{C}\to\mathsf{A}}$ expresses an interaction on the operation *auth* having C for sender and A for receiver.

A choreography can also be used to model a multiparty interaction, and indeed this is its most common use. As an example, we can describe the behaviour of the client role (C), the ATM role (A) and the bank role (B), discussed in Section 4, at least for the *balance* and *withdraw* operations.

$$auth_{\mathsf{C}\to\mathsf{A}}; \big(withdraw_{\mathsf{C}\to\mathsf{A}}; amount_{\mathsf{C}\to\mathsf{A}}; getAmount_{\mathsf{A}\to\mathsf{B}}; provAmount_{\mathsf{B}\to\mathsf{A}}; \\ (dispense_{\mathsf{A}\to\mathsf{C}} + overdraft_{\mathsf{A}\to\mathsf{C}}) \\ + balance_{\mathsf{C}\to\mathsf{A}}; askAmount_{\mathsf{A}\to\mathsf{B}}; repAmount_{\mathsf{B}\to\mathsf{A}}; amount_{\mathsf{A}\to\mathsf{C}}\big)^*; quit_{\mathsf{C}\to\mathsf{A}}$$

A notion of *conformance* is then introduced as a relation among a local contract L, a multiparty contract H and a role R, formalising the possibility to implement the multiparty contract H by adopting a peer following the local contract L to realise the role R. For instance, a peer following the local contract

$$!auth; (!withdraw; !amount; (?dispense + ?overdraft) + !balance; ?amount)^*; !quit \qquad (2)$$

could be used to realise the client role C in the multiparty contract above. Note that local contracts are based on send and receive operations and indeed coincide with binary contracts (see Section 3.4).

The theories for multiparty contracts formalise the notion of conformance above. For instance, Bravetti and Zavattaro [2007] introduce a notion of correctness for implementations of choreographies based on the following intuition: a system is correct if, for every reachable state, it is always possible for the peers to reach a successful state. This notion of correctness assumes fairness, as it is always considered possible to exit from loops if this is necessary to reach a successful state. Under this assumption, it allows one to check whether the parallel composition of peers following some given local contracts is a good implementation of a choreography. A (sound but not complete) decidable characterisation of conformance is then obtained as a combination of the above notion of correctness and refinement of local contracts.

A theory of multiparty contracts based on this approach has been proposed by Bravetti and Zavattaro [2008b]; this theory has been subsequently extended by considering a stronger notion of correctness according to which output actions cannot wait indefinitely [Bravetti and Zavattaro 2009b], by considering asynchronous instead of synchronous communication [Bravetti and Zavattaro 2008a], and by taking a language independent approach by representing processes as labelled transition systems [Bravetti and Zavattaro 2009a]. A technique to generate local contracts for peers conformant to a given choreography based on a notion of projection similar to the one studied for multiparty session types has been developed by Lanese et al. [2008].

Castagna et al. [2012] present a choreography language (they call "global types" their choreographies, but how to actually type processes using them is not described) featuring also interactions with multiple targets, representing a multicast, and a shuffling operator to specify that two behaviours can be interleaved arbitrarily. The paper characterises which choreographies can be implemented without cover channels, that is by using only interactions explicit in the choreography.

An alternative graphical model for the specification of choreographies — *collaboration diagrams* — was proposed by Bultan and Fu [2007] where the notion of realisability corresponds to the possibility of correctly implementing a given collaboration diagram as a parallel composition of services.

4.4. Extensions of multiparty session types

We present below approaches that extend multiparty session types and the related theory in different directions.

Exceptions. Carbone [2009] extends the work on exceptions for binary session types (Section 3.5) to deal with multiparty interactions. The paper shows by way of an example how to project a choreography with exceptions to derive the description of single endpoints. This work has been refined by Capecchi et al. [2010], where asynchronous exceptions that can be thrown to any subset of the participants of a multiparty session are considered. Operators dealing with exceptions are also available in the Conversation Calculus [Vieira et al. 2008].

Assertions. Bocchi et al. [2010] describe an approach to add assertions about data values to multiparty session types. Assertions concern the content of the exchanged messages, the choice of sub-conversations to follow, and invariants on recursions. This approach is further described in Section 6.2.

Parametricity. Yoshida et al. [2010] extend multiparty session types with dependent types and primitive recursion, allowing one to describe systems which are parametric on the number of participants. Indices are used to identify participants, and a foreach construct is used to let the number of interactions depend on the parameters.

Deniélou and Yoshida [2011] give another extension of the π-calculus with multiparty sessions where it is possible for participants to dynamically join and leave a session.

Compositionality. Montesi and Yoshida [2013] extend [Carbone and Montesi 2013] to a calculus that supports compositionality of choreographies. The key feature of the approach consists in adding π-calculus sends and receives to the choreography language. This goes in the direction of conversation types, described in Section 4.2.

Message-Passing Interface (MPI). Multiparty session types are also used for typing MPI programs. Honda et al. [2012] extend the work on parametric multiparty session types to describe the interactions within high-performance computing (HPC) programs. This work includes primitives for expressing collective operations idiomatic in HPC programs, such as scatter, for distributing an array amongst the participants, or reduce, for computing an operation depending on values contributed by a group of participants, as well as collective choices and loops. Traditionally, the branch and select primitives are dual and involve a participant that offers a menu of choices, from which the other chooses one. In HPC programs the idiom is different and participants choose a (same) path based only on local information gathered from previous interactions. No specific communication is needed for selecting a branch.

Following Honda et al. [2012], López et al. [2015] introduce a dependent functional type constructor and a notion of refinement types on protocols. This way, protocols can be parametric, for instance, on the size of the problem, and restrictions can be imposed on the exchanged data. As an example, $\Pi p\colon \Pi size\colon \{n\colon \mathsf{nat} \mid n\%p = 0\}.\mathsf{scatter}(0, \mathsf{MPI\_FLOAT}, size)$ denotes a protocol parametric on the number of participants (p) and on the size of the problem ($size$) that scatters a float array in chunks of $size/p$ amongst its participants. For that to succeed, the size of the problem must be a multiple of the number of participants.

Ng and Yoshida [2014] define *Pabble*, a protocol description language with dependent types. The language can describe an overall interaction topology designed for a variable number of participants arranged in multiple dimensions. These parameterised protocols in turn automatically generate local protocols for type checking parameterised MPI programs for communication safety and deadlock freedom. The theory underlying Pabble guarantees the termination of endpoint projection and of type checking algorithms.

Synthesis. The approaches described until now are top-down: a global description is specified, and local descriptions are derived from it. Lange and Tuosto [2012] explore a bottom-up approach, by defining type systems that make it possible (under certain conditions) to synthesise a multiparty global type, given a collection of local session types that describe endpoint behaviours (that is, local types).

Choreographic programming. Carbone and Montesi [2013] use global types to type multiparty processes written in a choreography language, and then project the choreography language on an endpoint language which extends the π-calculus with multiparty sessions. A main feature of the approach is that communications which are not in a causal dependence can be swapped, even if syntactically written in a sequence. The choreography language has no parallel composition operator: parallel composition is implicit. The approach allows one to type (thus guaranteeing deadlock freedom) also multiparty processes not typable according to Honda et al. [2008].

5. EXTENSIONS TO TYPE THEORIES

This section introduces further extensions to the theory of behavioural types, namely subtyping and polymorphism for session types, and contract refinement.

5.1. Subtyping for session types

The first formulation of subtyping for binary session types is by Gay and Hole [1999], and an extended and refined presentation is available in [Gay and Hole 2005]. It extends the formulation of Pierce and Sangiorgi [1996] dealing with single inputs/outputs to the sequences of inputs/outputs described by session types. The idea is to define the subtype relation on binary session types coinductively using a definition reminiscent of that of the simulation preorder for transition systems. For non-recursive session types an inductive definition in the form of inference rules suffices. In this case, the inference rules for input and output are as follows.

$$\frac{T \leq U \qquad V \leq W}{?T.V \leq ?U.W} \qquad\qquad \frac{U \leq T \qquad V \leq W}{!T.V \leq !U.W}$$

Subtyping is given the usual meaning, namely that $T_1 \leq T_2$ indicates that any value of type T_1 can be safely used in a context in which a value of type T_2 is expected. With this intuition in mind, consider a channel x with type $?U.W$. A process that uses x can safely read values of type U and, after this, x has type W. Consider now a channel x' from which one can receive values of a more specialized type T (i.e., $T \leq U$) and that, after this, has type V where $V \leq W$. This x', which has type $?T.V$, can safely be used instead of x, since x' will not carry any values that x would not carry – and the same is the case after any input. Therefore the subtype relation should be covariant for input.

On the other hand, consider a process that uses a channel y with type $!T.V$. On this channel, the process can safely send a value of type T and, after that, y has type V. If we have a channel y' that can be used for sending values of a more specialized type U (i.e., $U \leq T$) and that afterwards has the more specialized type W, then nothing bad will happen if we use y instead of y', since any value of type U that can be sent using y'

can also be sent using y. For this reason the subtype relation should be contravariant for output.

Now, the context uses not T_1 but the dual of T_1, hence the contravariant/covariant inversion with respect to the λ-calculus. In summary: input operations $(?, \&)$ are covariant, and output operations $(!, \oplus)$ are contravariant. Continuations are always covariant.

Combining the usual subsumption rule with the typing rules for linear input/output of Section 3.3, one obtains the following rules.

$$\frac{\Gamma, x\colon T_2 \vdash P \qquad T_3 \leq T_1}{\Gamma, y\colon T_3, x\colon !T_1.T_2 \vdash x!y.P} \qquad \frac{\Gamma, x\colon T_2, y\colon T_3 \vdash P \qquad T_1 \leq T_3}{\Gamma, x\colon ?T_1.T_2 \vdash x?y.P}$$

Behavioural techniques for subtyping, based on *web contracts* for binary sessions (see Section 5.3), are developed by Barbanera and de'Liguoro [2010]. This approach aims at a semantic characterisation of the notion of *sub-contract* in a language of session behaviours, which can be understood as behavioural types expressed in a process language. The sub-contracts are inspired by Castagna et al. [2009a]. The thus obtained subtyping relations are sound with respect to the original session subtyping of Gay and Hole [2005], in contrast to the fair subtyping discussed next.

The work on *fair subtyping* by Padovani [2011] extends the notion of subtyping from dyadic to (higher-order) multiparty sessions and also follows an approach based on contracts. In this case, too, the approach is inspired by behavioural techniques for processes (fair testing pre-order) similar to those adopted for multiparty contracts and discussed in Section 5.3. A challenging aspect of this work is the treatment of subtyping for (potentially) infinite sessions in conjunction with a *fairness* guarantee. Simply put, fairness here means that liveness and termination are preserved by subtyping. A syntactic axiomatisation and algorithms to decide fair subtyping are obtained.

In the quest for more flexible compositions of processes that retain the required safety properties, another kind of subtyping has been proposed by Mostrous and Yoshida [2009] for binary sessions, and by Mostrous et al. [2009] for multiparty sessions. This is based on the *re-ordering* of communications within a session, rather than on the possibility of sending and receiving different types. For such reorderings to be meaningful, communications need to be buffered, which means that input is *non-blocking* (or *asynchronous*). Because of this, it becomes possible to send values (and choice labels) in advance of inputs (or branchings), which opens significant possibilities for optimisation. A typical situation is when a type records that a value is to be sent after one or more input actions, and the implementation does not introduce a causal dependency between the input and the subsequent outputs: this form of subtyping allows a process to send such outputs in advance, providing for more efficient communication. The intuitive idea can be understood by a simple example. Channel x in process $x?y.x!5$ may be assigned a type $!\mathsf{nat}.?\mathsf{bool}.\mathsf{end}$ (a supertype of type $?\mathsf{bool}.!\mathsf{nat}.\mathsf{end}$), allowing the process to safely interact with process $x?z.x!\mathsf{true}$.

5.2. Polymorphism for session types

The first study of polymorphic sessions and specifically bounded polymorphism is by Gay [2008]. This work combines subtyping and polymorphism in the style of system F with subtyping, $\mathsf{F}_{<:}$. In particular, the usual branching and selection session types are extended with a payload type that also specifies a bound, leading to types of the shape below, where type variables X_i are bounded by types T_i and may appear in the U_i.

$$\&\{l_i(X_i \leq T_i)\colon U_i\}_{i\in I} \qquad \oplus\{l_i(X_i \leq T_i)\colon U_i\}_{i\in I}$$

The above type is assigned to terms of the shape $x \rhd \{l_i(X_i \leq T_i)\colon P_i\}_{i\in I}$ for branching, and to $x \lhd l(B).P$ for selection. As can be seen, type instantiation is "piggybacked"

into selection and branching. This provides for a simpler language, avoiding additional constructs.

The above work is adapted to an object-oriented setting by Dezani-Ciancaglini et al. [2006]. A notable aspect of this work is that choice (selection and branching) is not based on labels but rather it is guided by the class of a communicated object, which leads to a better integration with the object-oriented paradigm. This makes bounded polymorphism more challenging because subtyping can introduce ambiguity in the type-driven choice of a branch. Goto et al. [2015] define a polymorphic system for multiparty sessions in the style of contracts, with the distinguishing feature that type instantiation can affect multiple participants. We should also mention that logical interpretations of sessions, detailed in the next section, introduce polymorphism with universal quantification as input and existential quantification as output of a type.

5.3. Refinement for contracts

A notion corresponding to subtyping appears in the study of contracts, namely that of *refinement*. Using an adapted testing-based equivalence, Castagna and Padovani [2009] provide a semantic account of how contracts can be related, in terms of the final outcome (deadlock, success or indefinite progress) of every sub-component involved in the contracts.

As in [Castagna et al. 2009b], the refinement relation defined on contracts as formalized by Padovani [2010a] allows for safe replacement of services. The refinement relation coincides with the well-known *must testing* preorder [De Nicola and Hennessy 1984]; this was proved by Bernardi [2013], which also introduces refinements for clients. Bravetti and Zavattaro [2008b] consider the impact of fairness on contract refinement, showing a reduction of the notion of contract refinement to *should testing* [Rensink and Vogler 2007] in place of must testing. Fairness is useful in case of infinite behaviour in which it is necessary to assume the possibility to exit from loops in order to guarantee completion. For instance, fairness needs to be considered to prove that the contract

$$!auth; (!withdraw; !amount)^*; !quit$$

refines the client behaviour of contract (2) in Section 4.3. In fact, a peer following this restricted behaviour (no balance requests could be issued) can be used to implement the choreography presented in Section 4.3 because correctness continues to be guaranteed.

Barbanera and de'Liguoro [2010] take inspiration from the work on refinement for contracts and use it to give a new account of the behavioural semantics of session types, using the notions of compliance and sub-behaviour from the work on contracts. Bernardi and Hennessy [2013] show that the refinement relation for servers equals the must testing preorder only if contracts are finite state, and that the refinements for clients coincide only in languages as restricted as the finite part of session behaviours of Barbanera and de'Liguoro [2010].

6. LOGICS

This section introduces a linear logic interpretation of session types, and different works on the logical refinement of session types and behavioural contracts.

6.1. Linear logic foundations of session types

Linearity is an important and recurring theme in concurrency and, in particular, in (behavioural) type systems for process calculi; already Honda [1993] mentions linear logic as a source of inspiration for some aspects of session types. Caires and Pfenning [2010] introduce a Curry-Howard style interpretation of binary session types

in intuitionistic linear logic that exposes a deep correspondence between linear logic propositions and session types. The correspondence faithfully interprets the communication discipline of session-typed processes as reductions in logical derivations and conversely, as for the Curry-Howard isomorphism.

The basic correspondence between linear logic propositions and session types as described in Section 3.2 is as follows.

Proposition	Session type	
$T \multimap S$	$?T.S$	(input)
$T \otimes S$	$!T.S$	(output)
$S \,\&\, T$	$S\&T$	(choice offer)
$S \oplus T$	$S \oplus T$	(choice selection)
$\mathbf{1}$	`end`	(termination)
$!S$	$\hat{}S$	(shared channel type for sessions of type S)

In traditional functional interpretations of (intuitionistic) linear logic, an object of type $A \multimap B$ is a linear function that, when given an argument of type A, returns a result whose type is B [Girard and Lafont 1987]. In the interpretation an object of type $x{:}A \multimap B$ implements on channel x a session that first receives on x a session (channel) of type A, and afterwards behaves as B. Here B specifies a continuation session behaviour on x that somehow relies on the input session. These basic ideas can be explained by looking at the typing rules. Under the intuitionistic system processes are typed using judgements of the form

$$\Gamma; \Delta \vdash P :: z{:}A$$

Here Γ and Δ are typing contexts: Γ declares the shared channels (subject to contraction and weakening) while Δ declares the session channels (subject to the strict linear discipline). The $z{:}A$ on the right is a singleton typing context, declaring exactly a distinguished session. The judgement above may be naturally read as: process P when composed with shared servers complying with Γ and (open) sessions complying with Δ will safely provide a session of type A at channel z. The rules for output and input are as follows.

$$\frac{\Gamma; \Delta \vdash P :: y{:}A \qquad \Gamma; \Delta' \vdash Q :: x{:}B}{\Gamma; \Delta, \Delta' \vdash (\nu y)x!y.(P \mid Q) :: x{:}A \otimes B} \qquad \frac{\Gamma; \Delta, y{:}A \vdash P :: x{:}B}{\Gamma; \Delta \vdash x?y.P :: x{:}A \multimap B}$$

Note that the continuation process in the output case mentions two sub-processes P and Q, where Q is the session continuation process (on x) while P is the process that implements the session channel output in the communication (on y). This formulation subsumes the usual rule for output, given that P can just act as a forwarder process (implementing the identity or copy-cat session), so that the rule describes *bound output* as in the internal mobility discipline introduced by Sangiorgi [1996].

Process composition is typed by a cut rule, which combines parallel composition and channel restriction.

$$\frac{\Gamma; \Delta \vdash P :: x{:}A \qquad \Gamma; \Delta', x{:}A \vdash Q :: T}{\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: T}$$

It is useful to consider a simple example. We describe a client process that wishes to deposit money to a bank account via an ATM machine. The client does so by sending to the ATM her authentication information, after which she may send the amount she wishes to deposit. The ATM will then send back a receipt of the operation. From the point of view of the client, the session protocol followed by the ATM can be described by the following type.

$$\mathsf{ATMProto} \triangleq \mathsf{auth} \multimap \mathsf{amount} \multimap (\mathsf{receipt} \otimes \mathbf{1})$$

Here auth, amount and receipt are types that represent shareable values of basic data types (e.g. strings and integers). If we assume that s is the session channel along which the client and the ATM interact, the following process implements the client:

$$\mathsf{BCIntBody}_s \triangleq s!id.s!n.s?r.\mathbf{0}$$

The process above specifies a client that first sends her authentication information id, then the amount n to be deposited and finally receives the appropriate receipt. The following judgement is derivable.

$$\cdot; s{:}\mathsf{ATMProto} \vdash \mathsf{BCIntBody}_s :: -{:}\mathbf{1}$$

The ATM code is as follows:

$$\mathsf{ATMBody}_s \triangleq s?auth.s?amt.s!rc.\mathbf{0}$$

By composing the two processes with the cut rule, we obtain the following:

$$\cdot; \cdot \vdash (\nu s)(\mathsf{ATMBody}_s \mid \mathsf{BCIntBody}_s) :: -{:}\mathbf{1}$$

It is also possible to develop the interpretation on top of a classical linear logic formulation [Wadler 2012; Caires et al. 2015]. The intuitionistic formulation seems particularly intuitive, notwithstanding the non-standard look of typing rules, at least when compared with traditional session type systems.

The basic interpretation can be developed in many ways, and applied in several interesting settings. Toninho et al. [2011] and Pfenning et al. [2011] enrich the type system based on pure linear logic with dependent types and modalities to control how much information is communicated and show how the resulting framework can express proof-carrying code certified with digital signatures in a logically motivated way. Toninho et al. [2012] also introduce typed encodings of the simply typed λ-calculus into session-typed π-calculus motivated by the linear logic interpretation. Interestingly, one of such encodings corresponds to parallel evaluation (futures). DeYoung et al. [2012] show how a slight modification of the logical interpretation is enough to represent session-typed processes with asynchronous (or buffered) communication.

Wadler [2012] establishes a connection between the presentation of session types of Gay and Vasconcelos [2010] and linear logic, and shows how a simple modification yields a process calculus free from deadlock; the deadlock freedom is a consequence of the correspondence with linear logic. Caires et al. [2013] develop a complete theory of polymorphic session types based on second order linear logic, which for the first time dissects the notion of behavioural polymorphism. Key technical results include session fidelity and global progress, and remarkably also relational parametricity, which is useful for reasoning about information hiding (in terms of hiding of local protocols). Toninho et al. [2013] study a monadic integration of a functional language and a process language with session types, allowing one to express general higher-order session-typed processes.

6.2. Logically refined session types and behavioural contracts

In the setting of binary session types, Baltazar et al. [2012b] develop a notion of refined session types using the multiplicative linear logic as the language of refinements. The process language extends the π-calculus with assume and assert commands that guide the refinements, allowing for fine-grained specifications of communication protocols in which refinement formulae are seen as logical resources rather than persistent truths. This work can be seen as a generalisation of the works on type and effect systems for correspondence assertions of Gordon and Jeffrey [2003] and Bonelli et al. [2005].

In the setting of multiparty sessions, Bocchi et al. [2010] blend the theory of global types with a design-by-contract approach. In particular, this approach introduces

global and local types where data are explicitly added to interactions and used to specify in a suitable logic assertions on the communicated values and invariants in recursions. The assertions are written in a classical first-order logic. The following shows an example of a global type with assertions.

$$
\begin{aligned}
&\mathsf{A} \to \mathsf{B}\colon \{x\colon \mathsf{int} \mid x > 0\} \\
&\mathsf{B} \to \mathsf{C}\colon \{ \\
&\qquad \{x \geq 5\}\ ge : \mathsf{C} \to \mathsf{B}\colon \{y\colon \mathsf{int} \mid y\%2 = 0\}, \\
&\qquad \{x < 5\}\ lt : \mathsf{B} \to \mathsf{C}\colon \{z\colon \mathsf{bool} \mid z \iff x = 4\} \\
&\quad \}
\end{aligned}
$$

The three participants A, B, and C follow the protocol described by the interactions in the global type, but, unlike other approaches, interactions also establish constraints that must hold for the data that are communicated. First, A sends to B a value x that must be strictly positive. Participants B and C then engage in a choice operation, governed by labels ge and lt; the actual choice depends on x being greater or equal to 5. Finally, if choice gt is selected (by B), participant C sends an even number back to B, otherwise it receives from B the result of the evaluation of $x = 4$.

Global assertions introduce two issues in the definition of projection:

— global types cannot be projected when one of the senders is not able to fulfill its obligations because of 'lack of information', and
— the choices that a participant makes should not ruin later choices made by other participants.

The first of these issues can be dealt with by restricting the attention to *history sensitive* global types. These are types such that, for each interaction, the sending participant knows all the (free) variables found in the assertion associated with the interaction. The second issue will not occur for *temporally satisfiable types*. These are types such that for each possible set of values that satisfy an assertion ϕ and for all assertions ψ that occur later, there exists a set of values that satisfy ψ. Decidability of the assertion logic enables certain positive results. Firstly, history sensitivity and temporal satisfiability are decidable and preserved by the projection operation. Secondly, it is possible to validate annotated processes (Bocchi et al. [2010] use a version of the π-calculus with assertions) against local assertions. Finally, well-typed annotated processes are error free.

Bartoletti et al. [2012a] use contracts at run-time to allow participants to interact. There, a participant declares its contract independently of the others and then advertises it; compatible advertised contracts can then be stipulated to form a multiparty agreement. This agreement establishes a session within which the participants of the stipulated contracts interact by performing the actions dictated by the agreement. Bartoletti et al. [2012b] study the computational aspects of the framework in [Bartoletti et al. 2012a]. A type system for ensuring honesty has been given by Bartoletti et al. [2013], while Bartoletti et al. [2015] give a contract model based on multiparty session types for the framework in [Bartoletti et al. 2013]. A methodology for designing and composing services such that security policies are enforced locally is given by Bartoletti et al. [2008]. Safety properties are specified in contracts and a *call-by-contract mechanism* enforces them at composition time.

7. CLASSES OF BEHAVIOURAL PROPERTIES

All type systems aim to capture a specific property for programs written in a particular language. For instance, the type systems in Section 6.1 guarantee progress by

construction. However, the study of a particular property is sometimes the main motivation for understanding particular behavioural features.

The study of program properties usually distinguishes between safety and liveness properties. A safety property expresses that an undesirable program event will never happen during a program execution (or, equivalently, the invariant property of the undesirable event always being absent), whereas a liveness property describes that a desirable program event will occur eventually during the execution of the program. Type systems have traditionally been well suited for expressing safety properties; a particular challenge has been how to use behavioural types also to express liveness properties. We recall below the main safety properties ensured by the approaches described until now, and then we describe some approaches aimed at liveness properties.

7.1. Safety properties

In all the approaches we have discussed, well-typed programs are exempt from a series of common programming errors and, in general, enjoy various desirable properties.

First of all, interactions are free from *communication errors*. A classical example of such an error is a mismatch in the type of an exchanged message, as in $x^+!5 \mid x^-?y.(\mathsf{if}\ y\ \mathsf{then}\ldots\mathsf{else}\ldots)$ where one process sends an integer but the receiver expects a boolean. Other communication errors arise when two parallel processes try to interact on a given channel in a non-compatible way, for instance by performing two outputs $x^+!5 \mid x^-!7$, or a selection and an input $x^+ \triangleleft \mathit{quit} \mid x^-?y$. In general, the duality constraint that relates the session types associated with two peers of a session channel ensures that communication is *half duplex*, namely that at no time the interacting processes simultaneously send messages to each other. Limited forms of *full-duplex* communication can be achieved by means of relaxed subtyping relations [Mostrous et al. 2009].

Because of linearity constraints, communications on session channels are guaranteed to be *race free*. An example of race is given by two processes competing for a given resource, e.g., two outputs competing for a same input $x^+!5 \mid x^+!7 \mid x^-?y$. In turn, race freedom implies an interesting *confluence* result on session communications that is directly related to partial confluence discussed in [Kobayashi et al. 1999]. These properties are particularly relevant since they ensure the deterministic outcome for a whole family of concurrent computations.

The approaches discussed so far also guarantee that communications inside a single session do not block. However, deadlocks involving more than one session are possible in many of them. This is not the case for contracts, since only one session is considered. This is also not the case for the systems discussed in Section 6.1, where deadlock freedom is a consequence of the properties of the logic. Additionally, when interactions are guaranteed to be finite, deadlock freedom coincides with lock freedom [Kobayashi 2002], a liveness property ensuring that each pending communication eventually completes (under a fair scheduling). A system where a participant diverges without ever reaching a state where a given action a is enabled is deadlock free, since it never gets stuck, but not lock free, since the action complementary to a never gets executed.

7.2. Channel activeness/responsiveness

In communication-centred applications such as web services or distributed protocols, it is important that every request from a client is handled by a server. From the client's point of view, it is important that every valid request gets handled eventually by the server and moreover, that the client eventually obtains an answer. From the server's point of view, it is important that whenever a request is received, the client will respect the communication protocol.

This notion has been dealt with using behavioural types. Acciai and Boreale [2008] define the usage of a channel r to be *responsive* if a communication on r is guaranteed to happen eventually. Gamboni and Ravara [2010] call this property *activeness* and instead define responsiveness as an additional property. According to them, a channel endpoint c is *active* in the process P, if P is guaranteed to eventually perform an input on c. The endpoint c is instead said to be *responsive* if, every time the process receives (respectively, sends) a message on that channel, it is guaranteed to be active and responsive on the channels received (respectively, sent) via c, in the terms specified by a *channel type* of a process.

Acciai and Boreale [2008] define a type system for guaranteeing responsiveness, using a combination of techniques for deadlock and livelock avoidance together with ones used for describing linearity and receptiveness [Sangiorgi 1999]. The setting is a monadic synchronous π-calculus. The idea of the type system is to build a dependency graph whose vertices are responsive names of processes. In this graph, there is an edge from name a to name b if an output action involving a is dependent upon an input action on the name b. The type system then checks if the dependency graph is acyclic. Gamboni and Ravara [2010] work with the full synchronous polyadic π-calculus; in this case, the type system uses a notion of process types that specialise channel types to represent the interface between a process and its environment. The type algebra covers spatial, logical, and dynamical aspects of process types [Gamboni 2010], and the causal relations between channel usages are captured by *behavioural statements* embedded in process and channel types. These express the usage of channel endpoints between a process and its environment.

7.3. Capturing properties using spatial types

To achieve a proof system for properties such as race freedom, unique receptiveness [Sangiorgi 1999] and deadlock freedom, Acciai and Boreale [2010] describe a type system for the π-calculus that uses notions from spatial logic as well as a notion of behaviour. Names bound by restriction are typed with formulae from a "shallow" spatial logic (talking only about the next possible action), and processes are typed with terms from a CCS-like process calculus.

The type system allows model checking of spatial formulae. The importance of spatial logic in this setting is that the logical formulae impose constraints on the permissible spatial structure of processes; the structure of a π-calculus process and its type will be essentially the same. The class of properties that can be captured using this approach includes safety properties and some liveness properties.

7.4. Termination and deadlock freedom

The notions of termination and deadlock freedom are central in the theory of concurrent processes: non-termination is sometimes desirable—for instance we would not want an operating system to terminate—and in other settings termination must be ensured—requests in service-oriented applications should clearly be fulfilled.

Yoshida et al. [2004] and Berger et al. [2005] use the π-calculus to encode the simply typed λ-calculus. The goal is to show strong normalisation for this calculus by means of the combination of a π-calculus type system that will provide a sound characterisation of strong normalisation and a typed version of Milner's encoding of the λ-calculus. In this work, type judgments are of the form $\Gamma \vdash P \rhd A$, where A is an *action type*. An action type should be thought of as a finite directed graph whose vertices are names annotated with an input/output polarity and whose edges describe the causal input/output dependencies between names. The underlying idea of the type system is to ensure strong normalisation by ensuring that action types do not have cyclic dependencies between inputs and outputs and that inputs and outputs alternate.

Kobayashi [1998] uses a similar notion of causality in the form of tag orderings and graph types to prove deadlock freedom. Kobayashi [2002] uses behavioural types very similar to session types to reason about global properties of systems, in particular lock freedom. Termination of processes has also been tackled using conventional type systems; see e.g., the works by Sangiorgi [2006] and Demangeon et al. [2010]. Type systems of this kind have also been employed to ensure deadlock freedom and lock freedom (the latter, as already said, is the property that under a fair scheduling each pending communication eventually completes), in a series of papers by Kobayashi and coauthors, e.g., [Kobayashi 1998; Kobayashi 2005]. Different type systems may also be combined; e.g., the most powerful system for lock freedom [Kobayashi and Sangiorgi 2010] combines those for deadlock freedom and termination. Another relevant application area for these types has been security; see e.g., the system proposed by Haack and Jeffrey [2005] for secrecy and authenticity.

7.5. Progress for session type systems

An important kind of liveness property is that of progress for sessions: throughout a session, every process involved will never get stuck, since for every top-level input (respectively output) there will eventually appear a matching top-level output (respectively input). The notion of progress is close to the notion of lock freedom [Kobayashi 2002], however the former has been defined for session calculi, while the latter for π-calculus, hence the two are not easy to compare. A discussion about this is presented in [Padovani 2013b].

Session type systems can assure a local progress property *within a single session*, but they fall short in assuring progress when several (possibly multiparty) sessions are interleaved with each other. This follows from the fact that each session is typed in isolation, and the session type associated with a session endpoint is usually unrelated with the session types of session endpoints that are interleaved with it. More refined type systems that assure the progress property in presence of interleaved sessions are given by Dezani-Ciancaglini et al. [2007] for synchronous binary sessions and by Coppo et al. [2016] for asynchronous multiparty sessions. The basic idea of these type systems is to keep track of the dependencies between different sessions: a dependency $a \prec b$ indicates that there is an input action performed on a session opened on the service name a that blocks some other action performed on sessions opened on the service name b. Progress is guaranteed provided that $\prec$ is acyclic. This dependency-based mechanism is quite conservative and there exist practically relevant session patterns that yield circular dependencies but have progress. In particular, *nested sessions*, whereby all the input actions pertaining the session are completely nested within the actions of other sessions, have progress even if involved in circular dependencies. For this reason, the type system by Coppo et al. [2016] discriminates services whose sessions are nested and tolerates circular dependencies involving only them without compromising progress. Identifying session dependencies and properly classifying services requires a fair amount of type information associated with processes. Coppo et al. [2013] provide an inference algorithm for the type system.

All the above type systems that capture progress use whole sessions as units for determining dependencies between services. This means that circular dependencies are introduced by interleaving of sessions that block each other on input actions at different stages of their evolution; such circular dependencies render many processes with progress ill typed. Another consequence is that these type systems impose very restrictive constraints on session delegation. To overcome these limits, Padovani [2013b] and Vieira and Vasconcelos [2013] propose more refined type systems where dependency information concerns the single actions described in session types, rather than whole sessions. Padovani [2014] proposes a type system ensuring deadlock and lock freedom

of linear π-calculus processes. Thanks to the encoding of binary sessions into the linear π-calculus [Dardha et al. 2012], this type system can be used for reasoning about progress of interleaved binary sessions with better accuracy compared to [Padovani 2013b]. All of these works have been inspired from the type system for lock-free processes by Kobayashi [2002]. Finally, the systems described in Section 6.1 also ensure progress, while that in Section 7.3 ensure progress on the client side.

8. RELATING APPROACHES

Given the quantity and variety of approaches to session types and behavioural contracts highlighted by the previous sections, an obvious question to ask is how the various notions are related. This includes the study of how approaches to session types and behavioural contracts relate to conventional types and to communicating automata.

The relation between different approaches to session types and behavioural contracts has been studied by a number of authors. Bernardi and Hennessy [2012] use (a subset of) contracts by Castagna et al. [2009b] to define a fully abstract model of session types ordered by their subtyping relation. Bernardi [2013] shows that the same model can be defined in terms of must testing refinements for services and clients and extends the model to higher-order session types. While session types can be embedded in contracts, the existence of the converse embedding is still an open question.

Another issue that has been studied is whether session types can be captured using conventional type systems. Padovani [2010b] presents a session type system where choices are modelled using intersection and union types, and discusses the differences with the usual approach.

An encoding of session types into usage types extended with variant types is hinted at in [Kobayashi et al. 1996] and described in greater detail in the extended version of [Kobayashi 2003], but its properties are not discussed. Demangeon and Honda [2011] present a translation of binary session types into linear types for a polyadic π-calculus with directed choice and show that this translation is fully abstract. Dardha et al. [2012] show a translation from a π-calculus with binary session types to a π-calculus with standard linear, union and variant types. The translation is robust, as proved by extending it to deal with subtyping, polymorphism and higher-order features. Furthermore, it allows one to prove many results for session types as corollaries of corresponding results for standard types. Gay et al. [2014] present an encoding of a π-calculus with binary session types [Gay and Hole 2005] to a π-calculus with the generic type system [Igarashi and Kobayashi 2004] (thus, the target language has no extra features, contrary to the works just presented). The encoding deals with session type environments, polarities (which distinguish session channels endpoints), and labelled sums. They show forward and reverse operational correspondences for the encoding, as well as typing correspondences. Session subtyping, however, is only faithfully encoded if the target language also has record constructors (and the corresponding subtyping rules).

Hüttel [2011] proposes a general type system for ψ-calculi that also makes it possible to obtain type/effect systems as instances, including a version of the system of Gordon and Jeffrey [2003] for correspondence types. Hüttel [2013] proposes a similar approach to provide a general type system for a class of resource-aware type systems including both conventional type systems for linear names [Kobayashi et al. 1999] and the action types of Berger et al. [2005].

Communicating automata are finite state machines that communicate by exchanging messages via half-duplex channels (i.e. channels that provide communication in both directions, but only in one direction at a time). Gouda et al. [1984] show that a subclass of communicating automata composed by just 2 machines ensures freedom from deadlocks and from orphan messages. The first results about equivalence between

session types and communicating automata come from Villard [2011], which uses systems of communicating automata as contract specifications. Villard proves that the two-machine subclass of communicating automata characterises exactly binary session type behaviours. Deniélou and Yoshida [2013] explore this connection further in the multiparty case. Since a generalization of the notion of half-duplex does not work, they propose instead the definition of a multiparty compatibility property which allows for a sound and complete characterisation of the class of communicating automata that can be expressed by the language of multiparty session types [Honda et al. 2008].

9. ALGORITHMS

Central concerns for type systems are the problems of *type checking* and *type inference*. The former asks whether $\Gamma \vdash P$ holds, given a typing context Γ and a process P. The latter asks whether one can find a typing context Γ for given P such that $\Gamma \vdash P$ holds. Given that one checks or infers the typing context, one could speak of "typing checking" and "typing inference" instead of "type checking" and "type inference". Nevertheless, we stick to the latter terminology, since it is the one used in the literature on the topic.

Decidability of type checking is a relevant property for session types, however the topic is explicitly mentioned in a few papers only. One of the first works to explicitly describe a type checking algorithm is by Bonelli et al. [2005]. Giunti [2011] proposes a type checking algorithm for a version of session types for the π-calculus similar to that of Giunti and Vasconcelos [2016].

There is as yet little work on type inference for behavioural types. For linear type disciplines, Igarashi and Kobayashi [2000] describe a type inference algorithm for linear types in the π-calculus with subtyping. Kobayashi et al. [2000] describe a type inference algorithm for a type system that guarantees deadlock freedom; in this system types are process-like terms called usages. In the setting of session types, Mezzina [2008] presents an algorithm for type inference for a service calculus obtained as a simplification of SCC [Boreale et al. 2006]. Tasistro et al. [2012] describe a polymorphic type system for binary session types without recursion or branching/selection and provide an algorithm for type inference for this system. Imai et al. [2010] describe a strategy for type inference in a binary session type system in a version of the π-calculus with branching and selection. The underlying idea is to develop a type-safe representation in Haskell of session types and to use this together with Haskell type inference.

For type systems incorporating a notion of subtyping, type checking relies on subtyping being decidable. Therefore algorithms for deciding the subtype relation are particularly relevant. In the setting of binary session types, the first work in this direction is by Gay and Hole [2005], that defines subtyping for binary session types (see Section 5.1). This paper also presents algorithms for deciding subtyping and for performing type checking.

In the approach that uses processes as types, using a CCS-like type language leads to undecidability issues related to model checking and equivalence and preorder checking. This has been studied by Hüttel et al. [2009] who show that all preorders are undecidable even for a class of CCS processes with recursive definitions and parallel composition only, that is without restriction or communication. For the type systems using spatial logic studied by Acciai and Boreale [2010], the related model checking problem is undecidable even for a class of processes equivalent to Petri nets [Acciai et al. 2010].

In the setting of contract refinement, the decidability of service refinement [Padovani 2010a] follows from the decidability of the must testing preorder on finite-state processes [Cleaveland and Hennessy 1993]. The addition of the fairness assumption in the context of contracts was considered by Bravetti and Zavattaro [2008b; 2009b], where

a reduction of the proposed contract refinement relation to should testing (instead of must testing) is presented; an algorithm for checking conformance in this case is obtained by composing the presented reduction to the algorithm for checking should testing [Rensink and Vogler 2007].

In the setting of multiparty sessions, Padovani [2011] considers fair subtyping and presents a number of algorithms. First, an algorithm for deciding whether a type is viable: only viable types can occur as types of correct sessions. Second, algorithms for reducing a type to normal form and for deciding subtyping. The results are extended by Padovani [2013a] to deal with open session types and a coarser equivalence relation, and an algorithm for deciding open fair subtyping in time $\mathcal{O}(n^4)$ is presented.

Designing well-formed choreographies is not easy. Both the synchronisation part and the data part should satisfy some conditions. For the synchronisation part, Lanese et al. [2013] propose an algorithm to enforce the conditions described by Lanese et al. [2008]. For the data part, instead, Bocchi et al. [2012] propose three different algorithms for transforming inconsistent constraints on the communicated data as defined by Bocchi et al. [2010] into consistent ones. The paper also discusses their suitability, and sketches a methodology based on the proposed algorithms.

ACKNOWLEDGMENTS

The authors would like to thank the contributions of Jakob Rehof and Bernardo Toninho. This work was supported by FCT through LaSIGE Research Unit, ref. UID/CEC/00408/2013, and by the COST Action IC1201 BETTY (Behavioural Types for Reliable Large-Scale Software Systems).

REFERENCES

Martín Abadi and Luca Cardelli. 1996. *A theory of objects*. Springer. I–XIII, 1–396 pages.

Lucia Acciai and Michele Boreale. 2008. Responsiveness in process calculi. *Theoretical Computer Science* 409, 1 (2008), 59–93.

Lucia Acciai and Michele Boreale. 2010. Spatial and behavioral types in the pi-calculus. *Information and Computation* 208, 10 (2010), 1118–1153.

Lucia Acciai, Michele Boreale, and Gianluigi Zavattaro. 2010. On the relationship between spatial logics and behavioral simulations. In *FOSSACS (LNCS)*, Vol. 6014. Springer, 146–160.

Pedro Baltazar, Luís Caires, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. 2012a. A type system for flexible role assignment in multiparty communicating systems. In *TGC (LNCS)*, Vol. 8191. Springer, 82–96.

Pedro Baltazar, Dimitris Mostrous, and Vasco Thudichum Vasconcelos. 2012b. Linearly refined session types. In *LINEARITY (EPTCS)*, Vol. 101. Open Publishing Association, 38–49.

Franco Barbanera and Ugo de'Liguoro. 2010. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*. ACM, 155–164.

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and union types: syntax and semantics. *Information and Computation* 119, 2 (1995), 202–230.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment. *J. Symb. Log.* 48, 4 (1983), 931–940.

Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda calculus with types*. Cambridge University Press.

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. 2008. Semantics-based design for secure web services. *IEEE Trans. Software Eng.* 34, 1 (2008), 33–49.

Massimo Bartoletti, Julien Lange, Alceste Scalas, and Roberto Zunino. 2015. Choreographies in the wild. *Science of Computer Programming* 109 (2015), 36–60.

Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. 2013. Honesty by typing. In *FMOODS/FORTE (LNCS)*, Vol. 7892. Springer, 305–320.

Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. 2012a. Contract-oriented computing in CO2. *Sci. Ann. Comp. Sci.* 22, 1 (2012), 5–60.

Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. 2012b. On the realizability of contracts in dishonest systems. In *COORDINATION (LNCS)*, Vol. 7274. Springer, 245–260.

Martin Berger, Kohei Honda, and Nobuko Yoshida. 2005. Genericity and the pi-calculus. *Acta Inf.* 42, 2-3 (2005), 83–141.

Giovanni Bernardi. 2013. *Behavioural equivalences for web services*. Ph.D. Dissertation. Trinity College Dublin.

Giovanni Bernardi and Matthew Hennessy. 2012. Modelling session types using contracts. In *SAC*. ACM, 1941–1946.

Giovanni Bernardi and Matthew Hennessy. 2013. Compliance and testing preorders differ. In *SEFM Workshops (LNCS)*, Vol. 8368. Springer, 69–81.

Giovanni Bernardi and Matthew Hennessy. 2014. Using higher-order contracts to model session types (extended abstract). In *CONCUR (LNCS)*, Vol. 8704. Springer, 387–401.

Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. 2008. Session and union types for object oriented programming. In *Concurrency, Graphs and Models (LNCS)*, Vol. 5065. Springer, 659–680.

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR (LNCS)*, Vol. 6269. Springer, 162–176.

Laura Bocchi, Julien Lange, and Emilio Tuosto. 2012. Three algorithms and a methodology for amending contracts for choreographies. *Sci. Ann. Comp. Sci.* 22, 1 (2012), 61–104.

Eduardo Bonelli, Adriana B. Compagnoni, and Elsa L. Gunter. 2005. Typechecking safe process synchronization. In *FGUC (ENTCS)*, Vol. 138. Elsevier, 3–22.

Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. 2006. SCC: a Service Centered Calculus. In *WS-FM (LNCS)*, Vol. 4184. Springer, 38–57.

Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. 2008. Sessions and pipelines for structured service programming. In *FMOODS (LNCS)*, Vol. 5051. Springer, 19–38.

Gérard Boudol. 1997. Typing the use of resources in a concurrent calculus. In *ASIAN (LNCS)*, Vol. 1345. Springer, 239–253.

Mario Bravetti and Gianluigi Zavattaro. 2007. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition (LNCS)*, Vol. 4829. Springer, 34–50.

Mario Bravetti and Gianluigi Zavattaro. 2008a. Contract compliance and choreography conformance in the presence of message queues. In *WS-FM (LNCS)*, Vol. 5387. Springer, 37–54.

Mario Bravetti and Gianluigi Zavattaro. 2008b. A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae* 89, 4 (2008), 451–478.

Mario Bravetti and Gianluigi Zavattaro. 2009a. Contract-based discovery and composition of web services. In *SFM (LNCS)*, Vol. 5569. Springer, 261–295.

Mario Bravetti and Gianluigi Zavattaro. 2009b. A theory of contracts for strong service compliance. *Mathematical Structures in Computer Science* 19, 3 (2009), 601–638.

Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. 1995. Lotos specifications, their implementations and their tests. In *Conformance testing methodologies and architectures for OSI protocols*. IEEE Computer Society, 468–479.

Antonio Brogi, Carlos Canal, and Ernesto Pimentel. 2004. Behavioural types and component adaptation. In *AMAST (LNCS)*, Vol. 3116. Springer, 42–56.

Roberto Bruni and Leonardo Gaetano Mezzina. 2008. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *AMAST (LNCS)*, Vol. 5140. Springer, 100–115.

Tevfik Bultan and Xiang Fu. 2007. Specification of realizable service conversations using collaboration diagrams. In *SOCA*. IEEE Computer Society, 122–132.

Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. 2005. Choreography and orchestration: a synergic approach for system design. In *ICSOC (LNCS)*, Vol. 3826. Springer, 228–240.

Luís Caires. 2008. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theoretical Computer Science* 402, 2-3 (2008), 120–141.

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral polymorphism and parametricity in session-based communication. In *ESOP (LNCS)*, Vol. 7792. Springer, 330–349.

Luís Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *CONCUR (LNCS)*, Vol. 6269. Springer, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2015. Linear logic propositions as session types. *Mathematical Structures in Computer Science* (2015), 57 pages. http://journals.cambridge.org/article_S0960129514000218 To appear.

Luís Caires and João Costa Seco. 2013. The type discipline of behavioral separation. In *POPL*. ACM, 275–286.

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

Sara Capecchi, Elena Giachino, and Nobuko Yoshida. 2010. Global escape in multiparty sessions. In *FSTTCS (LIPIcs)*, Vol. 8. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 338–351.

Marco Carbone. 2009. Session-based choreography with exceptions. In *PLACES (ENTCS)*. Elsevier, 35–55.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured interactional exceptions in session types. In *CONCUR (LNCS)*, Vol. 5201. Springer, 402–417.

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*. ACM, 263–274.

Samuele Carpineti, Giuseppe Castagna, Cosimo Laneve, and Luca Padovani. 2006. A formal account of contracts for web services. In *WS-FM (LNCS)*, Vol. 4184. Springer, 148–162.

Cyril Carrez, Alessandro Fantechi, and Elie Najm. 2003. Behavioural contracts for a sound assembly of components. In *FORTE (LNCS)*, Vol. 2767. Springer, 111–126.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009a. Foundations of session types. In *PPDP*. ACM, 219–230.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2012. On global types and multiparty sessions. *Logical Methods in Computer Science* 8, 1, Article 24 (2012), 45 pages.

Giuseppe Castagna, Nils Gesbert, and Luca Padovani. 2009b. A theory of contracts for Web services. *ACM Trans. Program. Lang. Syst.* 31, 5, Article 19 (2009), 61 pages.

Giuseppe Castagna and Luca Padovani. 2009. Contracts for mobile processes. In *CONCUR (LNCS)*, Vol. 5710. Springer, 211–228.

Samir Chouali and Ahmed Hammad. 2011. Formal verification of components assembly based on SysML and interface automata. *ISSE* 7, 4 (2011), 265–274.

Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. 2010. Adapting component behaviours using interface automata. In *EUROMICRO-SEAA*. IEEE Computer Society, 119–122.

Rance Cleaveland and Matthew Hennessy. 1993. Testing equivalence as a bisimulation equivalence. *Formal Asp. Comput.* 5, 1 (1993), 1–20.

Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. 1999. Static safety analysis for non-uniform service availability in Actors. In *FMOODS (IFIP Conference Proceedings)*, Vol. 139. Kluwer, 371–386.

Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. 1997. A set-constraint-based analysis of actors. In *FMOODS*. Chapman & Hall, 1–16.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION (LNCS)*, Vol. 7890. Springer, 45–59.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science* 26 (2016), 238–302.

Mario Coppo and Alberto Ferrari. 1993. Type inference, abstract interpretation and strictness analysis. *Theoretical Computer Science* 121, 1&2 (1993), 113–143.

Luís Cruz-Filipe, Ivan Lanese, Francisco Martins, António Ravara, and Vasco Thudichum Vasconcelos. 2008. Behavioural theory at work: program transformations in a service-centred calculus. In *FMOODS (LNCS)*, Vol. 5051. Springer, 59–77.

Luís Cruz-Filipe, Ivan Lanese, Francisco Martins, António Ravara, and Vasco Thudichum Vasconcelos. 2014. The Stream-based Service-Centered Calculus: a foundation for service-oriented programming. *Formal Aspects of Computing* 26, 5 (2014), 865–918.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *PPDP*. ACM, 139–150.

Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. In *ESEC-FSE*. ACM, 109–120.

Rocco De Nicola and Matthew Hennessy. 1984. Testing equivalences for processes. *Theoretical Computer Science* 34 (1984), 83–133.

Robert DeLine and Manuel Fähndrich. 2004. Typestates for objects. In *ECOOP (LNCS)*, Vol. 3086. Springer, 465–490.

Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2010. Termination in impure concurrent languages. In *CONCUR (LNCS)*, Vol. 6269. Springer, 328–342.

Romain Demangeon and Kohei Honda. 2011. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR (LNCS)*, Vol. 6901. Springer, 280–296.

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *POPL*. ACM, 435–446.

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In *ICALP (2) (LNCS)*, Vol. 7966. Springer, 174–186.

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut reduction in linear logic as asynchronous session-typed communication. In *CSL (LIPIcs)*, Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 228–242.

Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. 2007. On progress for structured communications. In *TGC (LNCS)*, Vol. 4912. Springer, 257–275.

Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. 2006. Bounded session types for object oriented languages. In *FMCO (LNCS)*, Vol. 4709. Springer, 207–245.

Mariangiola Dezani-Ciancaglini, Furio Honsell, and Yoko Motohama. 2005. Compositional characterisations of *lambda*-terms using intersection types. *Theoretical Computer Science* 340, 3 (2005), 459–495.

Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. 2005. L-doos: a distributed object-oriented language with session types. In *TGC (LNCS)*, Vol. 3705. Springer, 299–318.

Joshua Dunfield. 2012. Elaborating intersection and union types. In *ICFP*. ACM, 17–28.

Joshua Dunfield and Frank Pfenning. 2003. Type assignment for intersections and unions in call-by-value languages. In *FoSSaCS (LNCS)*, Vol. 2620. Springer, 250–266.

Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. 2004. Stuck-free conformance. In *CAV (LNCS)*, Vol. 3114. Springer, 242–254.

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *PLDI*. ACM, 268–277.

Maxime Gamboni. 2010. *Statically proving behavioural properties in the π-calculus via dependency analysis*. Ph.D. Dissertation. Instituto Superior Técnico, Technical University of Lisbon.

Maxime Gamboni and António Ravara. 2010. Responsive choice in mobile processes. In *TGC (LNCS)*, Vol. 6084. Springer, 135–152.

Simon J. Gay. 2008. Bounded polymorphism in session types. *Mathematical Structures in Computer Science* 18, 5 (2008), 895–930.

Simon J. Gay, Nils Gesbert, and António Ravara. 2014. Session types as generic process types. In *EXPRESS/SOS (EPTCS)*, Vol. 160. 94–110.

Simon J. Gay and Malcolm Hole. 1999. Types and subtypes for client-server interactions. In *ESOP (LNCS)*, Vol. 1576. Springer, 74–90.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Inf.* 42, 2-3 (2005), 191–225.

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular session types for distributed object-oriented programming. In *POPL*. ACM, 299–312.

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50 (1987), 1–102.

Jean-Yves Girard and Yves Lafont. 1987. Linear logic and lazy computation. In *TAPSOFT(2) (LNCS)*, Vol. 250. Springer, 52–66.

Marco Giunti. 2011. A type checking algorithm for qualified session types. In *WWV (EPTCS)*, Vol. 61. Open Publishing Association, 96–114.

Marco Giunti and Vasco Thudichum Vasconcelos. 2016. Linearity, session types and the Pi calculus. *Mathematical Structures in Computer Science* 26 (2016), 206–237.

Andrew D. Gordon and Alan Jeffrey. 2003. Typing correspondence assertions for communication protocols. *Theoretical Computer Science* 300, 1-3 (2003), 379–409.

Matthew Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2015. An extensible approach to session polymorphism. *Mathematical Structures in Computer Science* (2015), 45 pages. http://journals.cambridge.org/article_S0960129514000231 To appear.

Mohamed G. Gouda, Eric G. Manning, and Yao-Tin Yu. 1984. On the progress of communications between two finite state machines. *Information and Control* 63, 3 (1984), 200–216.

Christian Haack and Alan Jeffrey. 2005. Timed spi-calculus with types for secrecy and authenticity. In *CONCUR (LNCS)*, Vol. 3653. Springer, 202–216.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*. William Kaufmann, 235–245.

C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR (LNCS)*, Vol. 715. Springer, 509–523.

Kohei Honda, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2012. Verification of MPI programs using session types. In *EuroMPI (LNCS)*, Vol. 7490. Springer, 291–293.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *ESOP (LNCS)*, Vol. 1381. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.

Hans Hüttel. 2011. Typed ψ-calculi. In *CONCUR (LNCS)*, Vol. 6901. Springer, 265–279.

Hans Hüttel. 2013. Types for resources in ψ-calculi. In *TGC (LNCS)*, Vol. 8358. Springer, 83–102.

Hans Hüttel, Naoki Kobayashi, and Takashi Suto. 2009. Undecidable equivalences for basic parallel processes. *Information and Computation* 207, 7 (2009), 812–829.

Atsushi Igarashi and Naoki Kobayashi. 2000. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation* 161, 1 (2000), 1–44.

Atsushi Igarashi and Naoki Kobayashi. 2004. A generic type system for the pi-calculus. *Theoretical Computer Science* 311, 1-3 (2004), 121–163.

Atsushi Igarashi and Hideshi Nagira. 2007. Union types for object-oriented programming. *Journal of Object Technology* 6, 2 (2007), 47–68.

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 2010. Session type inference in Haskell. In *PLACES (EPTCS)*, Vol. 69. Open Publishing Association, 74–91.

Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. 2005. *Web Services Choreography Description Language Version 1.0*. Technical Report. W3C. http://www.w3.org/TR/ws-cdl-10/.

Naoki Kobayashi. 1998. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.* 20, 2 (1998), 436–482.

Naoki Kobayashi. 2000. Type systems for concurrent processes: from deadlock-freedom to livelock-freedom, time-boundedness. In *IFIP TCS (LNCS)*, Vol. 1872. Springer, 365–389.

Naoki Kobayashi. 2002. A type system for lock-free processes. *Information and Computation* 177, 2 (2002), 122–159.

Naoki Kobayashi. 2003. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support*. LNCS, Vol. 2757. Springer, 439–453. Extended version available at http://www-kb.is.s.u-tokyo.ac.jp/~koba/papers/tutorial-type-extended.pdf.

Naoki Kobayashi. 2005. Type-based information flow analysis for the pi-calculus. *Acta Inf.* 42, 4-5 (2005), 291–347.

Naoki Kobayashi and C.-H. Luke Ong. 2009. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*. IEEE Computer Society, 179–188.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1996. Linearity and the pi-calculus. In *POPL*. ACM, 358–371.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947.

Naoki Kobayashi, Shin Saito, and Eijiro Sumii. 2000. An implicitly-typed deadlock-free process calculus. In *CONCUR (LNCS)*, Vol. 1877. Springer, 489–503.

Naoki Kobayashi and Davide Sangiorgi. 2010. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.* 32, 5, Article 16 (2010), 49 pages.

Dimitrios Kouzapas, Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2016. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science* 26 (2016), 303–364.

Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*. IEEE Computer Society, 323–332.

Ivan Lanese, Francisco Martins, Vasco Thudichum Vasconcelos, and António Ravara. 2007. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*. IEEE Computer Society, 305–314.

Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2013. Amending choreographies. In *WWV (EPTCS)*, Vol. 123. Open Publishing Association, 34–48.

Cosimo Laneve and Luca Padovani. 2007. The must preorder revisited. In *CONCUR (LNCS)*, Vol. 4703. Springer, 212–225.

Cosimo Laneve and Luca Padovani. 2013. An algebraic theory for web service contracts. In *IFM (LNCS)*, Vol. 7940. Springer, 301–315.

Julien Lange and Emilio Tuosto. 2012. Synthesising choreographies from local session types. In *CONCUR (LNCS)*, Vol. 7454. Springer, 225–239.

Edward A. Lee and Yuhong Xiong. 2004. A behavioral type system and its application in Ptolemy II. *Formal Asp. Comput.* 16, 3 (2004), 210–237.

Barbara Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841.

Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *OOPSLA*. ACM, 280–298.

Leonardo Gaetano Mezzina. 2008. How to infer finite session types in a calculus of services and sessions. In *COORDINATION (LNCS)*, Vol. 5052. Springer, 216–231.

Robin Milner. 1992. Funtions as processes. *Mathematical Structures in Computer Science* 2, 2 (1992), 119–141.

Robin Milner. 1993. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. NATO ASI Series, Vol. 94. Springer, 203–246.

Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100, 1 (1992), 1–40.

Fabrizio Montesi and Nobuko Yoshida. 2013. Compositional choreographies. In *CONCUR (LNCS)*, Vol. 8052. Springer, 425–439.

Dimitris Mostrous and Nobuko Yoshida. 2009. Session-based communication optimisation for higher-order mobile processes. In *TLCA (LNCS)*, Vol. 5608. Springer, 203–218.

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global principal typing in partially commutative asynchronous sessions. In *ESOP (LNCS)*, Vol. 5502. Springer, 316–332.

Mayur Naik and Jens Palsberg. 2005. A type system equivalent to a model checker. In *ESOP (LNCS)*, Vol. 3444. Springer, 374–388.

Elie Najm and Abdelkrim Nimour. 1997. A calculus of object bindings. In *FMOODS*. Chapman & Hall.

Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. 1999a. Guaranteeing liveness in an object calculus through behavioural typing. In *FORTE (IFIP Conference Proceedings)*, Vol. 156. Kluwer, 203–221.

Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. 1999b. Infinite types for distributed object interfaces. In *FMOODS (IFIP Conference Proceedings)*, Vol. 139. Kluwer, 353–369.

Nicholas Ng and Nobuko Yoshida. 2014. Pabble: parameterised Scribble for parallel programming. In *PDP*. IEEE Computer Society, 707–714.

Flemming Nielson and Hanne Riis Nielson. 1993. From CML to process algebras. In *CONCUR (LNCS)*, Vol. 715. Springer, 493–508.

Flemming Nielson and Hanne Riis Nielson. 1996. From CML to its process algebra. *Theoretical Computer Science* 155, 1 (1996), 179–219.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. I–XXI, 1–450 pages.

Oscar Nierstrasz. 1995. Regular types for active objects. In *Object-Oriented Software Composition*. Prentice Hall International, 99–121.

Luca Padovani. 2010a. Contract-based discovery of Web services modulo simple orchestrators. *Theoretical Computer Science* 411, 37 (2010), 3328–3347.

Luca Padovani. 2010b. Session types = intersection types + union types. In *ITRS (EPTCS)*, Vol. 45. Open Publishing Association, 71–89.

Luca Padovani. 2011. Fair subtyping for multi-party session types. In *COORDINATION (LNCS)*, Vol. 6721. Springer, 127–141.

Luca Padovani. 2013a. Fair subtyping for open session types. In *ICALP (2) (LNCS)*, Vol. 7966. Springer, 373–384.

Luca Padovani. 2013b. From lock freedom to progress using session types. In *PLACES (EPTCS)*, Vol. 137. Open Publishing Association, 3–19.

Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*. ACM, 72:1–72:10.

Frank Pfenning, Luís Caires, and Bernardo Toninho. 2011. Proof-carrying code in a session-typed process calculus. In *CPP (LNCS)*, Vol. 7086. Springer, 21–36.

Benjamin C. Pierce. 1991. *Programming with intersection types, union types, and polymorphism*. Technical Report CMU-CS-91-106. CMU.

Benjamin C. Pierce and Davide Sangiorgi. 1996. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science* 6, 5 (1996), 409–453.

Garrel Pottinger. 1980. A type assignment for the strongly normalizable lambda-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 561–577.

Franz Puntigam. 2001a. State inference for dynamically changing interfaces. *Comput. Lang.* 27, 4 (2001), 163–202.

Franz Puntigam. 2001b. Strong types for coordinating active objects. *Concurrency and Computation: Practice and Experience* 13, 4 (2001), 293–326.

Franz Puntigam and Christof Peter. 2001. Types for active objects with static deadlock prevention. *Fundamenta Informaticae* 48, 4 (2001), 315–341.

António Ravara, Pedro Resende, and Vasco Thudichum Vasconcelos. 2012. An algebra of behavioural types. *Information and Computation* 212 (2012), 64–91.

António Ravara and Vasco Thudichum Vasconcelos. 2000. Typing non-uniform concurrent objects. In *CONCUR (LNCS)*, Vol. 1877. Springer, 474–488.

Jakob Rehof. 2013. Towards combinatory logic synthesis. In *BEAT*. 47–58.

Arend Rensink and Walter Vogler. 2007. Fair testing. *Information and Computation* 205, 2 (2007), 125–198.

John C. Reynolds. 1997. Design of the programming language Forsythe. In *Algol-like Languages*. Birkhäuser Basel, 173–233.

Davide Sangiorgi. 1996. Pi-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science* 167, 1&2 (1996), 235–274.

Davide Sangiorgi. 1998. An interpretation of typed objects into typed pi-calculus. *Information and Computation* 143, 1 (1998), 34–73.

Davide Sangiorgi. 1999. The name discipline of uniform receptiveness. *Theoretical Computer Science* 221, 1-2 (1999), 457–493.

Davide Sangiorgi. 2006. Termination of processes. *Mathematical Structures in Computer Science* 16, 1 (2006), 1–39.

Robert E. Strom and Shaula Yemini. 1986. Typestate: a programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171.

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in Plaid. In *OOPSLA*. ACM, 713–732.

Alvaro Tasistro, Ernesto Copello, and Nora Szasz. 2012. Principal type scheme for session types. *International Journal of Logic and Computation* 3, 1 (2012), 34–43.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *PPDP*. ACM, 161–172.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2012. Functions as session-typed processes. In *FoSSaCS (LNCS)*, Vol. 7213. Springer, 346–360.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-order processes, functions, and sessions: a monadic integration. In *ESOP (LNCS)*, Vol. 7792. Springer, 350–369.

Antonio Vallecillo, Vasco Thudichum Vasconcelos, and António Ravara. 2006. Typing the behavior of software components using session types. *Fundamenta Informaticae* 73, 4 (2006), 583–598.

Vasco Thudichum Vasconcelos. 1994. Typed concurrent objects. In *ECOOP (LNCS)*, Vol. 821. Springer, 100–117.

Vasco Thudichum Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70.

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

Hugo Torres Vieira, Luís Caires, and João Costa Seco. 2008. The Conversation Calculus: a model of service-oriented computation. In *ESOP (LNCS)*, Vol. 4960. Springer, 269–283.

Hugo Torres Vieira and Vasco Thudichum Vasconcelos. 2013. Typing progress in communication-centred systems. In *COORDINATION (LNCS)*, Vol. 7890. Springer, 236–250.

Jules Villard. 2011. *Heaps and hops*. Ph.D. Dissertation. ENS Cachan.

Philip Wadler. 2012. Propositions as sessions. In *ICFP*. ACM, 273–286.

Nobuko Yoshida, Martin Berger, and Kohei Honda. 2004. Strong normalisation in the pi-calculus. *Information and Computation* 191, 2 (2004), 145–202.

Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. 2010. Parameterised multiparty session types. In *FOSSACS (LNCS)*, Vol. 6014. Springer, 128–145.

Nobuko Yoshida and Vasco Thudichum Vasconcelos. 2007. Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication. In *SecReT (ENTCS)*, Vol. 171(4). Elsevier, 73–93.

Received January XXXX; revised January YYYY; accepted January ZZZZ