A Reactive Interpretation of Session-Based Concurrency

Jaime Arias
University of Bordeaux, CNRS
LaBRI UMR, Inria, France

Mauricio Cano
University of Groningen,
The Netherlands

Jorge A. Pérez
University of Groningen,
The Netherlands

Abstract

In *communication-centric* software systems, distributed services concurrently interact by following precise protocols. One approach to certify protocol correctness uses *behavioral types* to abstract protocols and statically check interacting programs. Behavioral types have been widely studied for programming calculi such as the π-calculus, which lack constructs for specifying *reactive behavior*, an increasingly relevant feature in many real-life interaction scenarios.

In this paper, we explore the use of the *synchronous reactive programming* paradigm as a uniform foundation for communication-centric programs. We focus on *session types*, a widely studied class of behavioral types, and on *session-based concurrency*, its associated computation model. We present an interpretation (formal translation) of session-based concurrency into ReactiveML, a synchronous reactive language. Our approach uniformly integrates communication, and timed and reactive behavior in concurrent programs which are more natural and concise than π-calculus specifications. To our knowledge, this is the first implementation of sessions in a synchronous reactive programming language.

1. Introduction

The goal of this paper is to describe our ongoing work on the first reactive implementation of *session-based concurrency* [23], a type-based approach to correct communicating programs. Our implementation relies on a formal interpretation (translation) of concurrent processes into ReactiveML, a reactive extension of ML [29].

Triggered by trends such as service-oriented computing, the development of formal models of *communication-centric systems* has received much attention [23, 24, 33, 39]. Within programming languages and concurrency communities, the interest has been in (formal) approaches to the specification of the interfaces (or contracts) of communicating programs, and in static/dynamic verification techniques to ensure that programs conform to these interfaces. One of such approaches uses *behavioral types* [25] to specify structured communication protocols and type checking to enforce protocol conformance. While traditional data types classify values, behavioral types classify patterns of interaction. *Session types* [23] are a class of behavioral types that organize concurrent interactions into logical units called *sessions*. In session-based concurrency, the behavior of each channel in a program (an *endpoint*) is described by a session type that abstracts its communication behavior.

Session-based concurrency has been traditionally formulated as a type theory for the π-calculus [31]; building upon these foundations, a number of practical tools and languages for protocol programming have been proposed [2, 34–36]. To cope with aspects such as asynchronous communication, events, and time, extensions of session-based π-calculi have been proposed (see, e.g., [6, 26]). While each of these extensions is suitable on its own, it is unclear how to integrate their valuable features in a uniform framework. Such an integration appears indispensable nowadays, as communicating software artifacts increasingly feature more complex forms of interaction. Consider, e.g., business protocols, which are naturally specified using sessions: typically, such protocols are influenced by both *contextual information* (e.g., stock availability) and *time constraints* (e.g., deadlines and timeouts), which may in turn trigger as *events* various *error-handling* and *adaptation* policies. All these concerns may affect protocol execution; a protocol may decree a run-time reconfiguration (say, to deploy a different protocol) in case of, e.g., missing information and/or an unmet deadline.

Combined, the difficulty of conceiving a comprehensive session-based framework and the complexity of current scenarios of structured communications cast doubts as whether the π-calculus is an adequate, sufficient basis for session-based concurrency. In this work, we describe a fresh approach towards such a comprehensive framework of communication-centric programming. We explore how programming calculi and languages based on the *synchronous*

REBLS'16, November 1, 2016, Amsterdam, The Netherlands.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

reactive programming paradigm (SRP) [4, 22, 30] can offer a uniform basis for session-based concurrency.

SRP is an event-based model of computation optimized for programming reactive systems. Synchronous languages are based on the *hypothesis of perfect synchrony*: reactive programs respond instantaneously and produce their outputs synchronously with their input. A synchronous program is meant to deterministically react to events coming from the environment: in essence, it evolves through an infinite sequence of successive reactions indexed by a global logical clock. During a reaction, each system component provides new output values based on the input values and on its internal state; the communication of all events between components occurs synchronously during each reaction. Thus, real physical time is not involved. Reactions are required to converge and computations are entirely performed before the current execution instant ends and the next one begins. This notion of time enables SRP programs to have an *order* in the events of the system, which enables reasoning about time-related properties [16, 19]. This way, e.g., one may verify whether a sequence of events is executed in the intended order (an elementary concern for session protocols) and whether a certain event (or sequence of events) is executed within t time units.

Our work aims at establishing to what extent SRP is a suitable model for session-based concurrency. In this paper, we report our main achievements so far, which can be summarized as follows:

- Exploiting ReactiveML and its formal semantics, we present an interpretation (formal translation) of session-based concurrency in SRP. This interpretation enjoys an important operational correspondence property, and enables a natural modeling of issues which are unnatural or hard to represent in models based on the π-calculus, such as e.g., event-based communication.
- We show that our interpretation of sessions can be implemented in ReactiveML. Hence, unlike most foundational developments based on the π-calculus, our work has a direct connection to practical programming. We illustrate our implementation using compelling examples where the combination of session-based concurrency and SRP can prove useful in practice.

Our interpretation paves the way for embedding communication-centric code into ReactiveML programs. Indeed, we can now use session-based constructs as atomic "blocks" within larger ReactiveML programs, therefore achieving a uniform account of structured communication and reactive, timed behavior. Our operational correspondence result ensures that such communication blocks coexist consistently with ReactiveML programs.

Next, we further illustrate our approach and contributions (§2). We then introduce both session-based concurrency and ReactiveML (§3). In §4 we present our interpretation; we illustrate it via examples (§5). We conclude by discussing directions of current/future work and by commenting on related works (§6). An online appendix contains omitted definitions and proofs [3].

2. Overview of Approach and Contributions

A Running Example. To illustrate session-based concurrency and our approach we use the *Buyer-Seller-Shipper protocol* [17]. This protocol concerns the interaction of three participants (Buyer, Seller and Shipper) involved in a transaction structured as follows:

1. Buyer requests an item from Seller.
2. Seller replies back asking for Buyer's unique address.
3. Buyer sends his address to Seller, confirming the order.
4. Seller forwards Buyer's address to Shipper.
5. Shipper sends to Buyer the estimated delivery time.
6. Buyer tells Shipper he is available to receive the item.

Using the constructs of session types we may unambiguously express a wide range of structured protocols. The session type $!A.T$ (resp. $?A.T$) is associated to a channel that first sends (resp. receives) a value of type A and then executes protocol T. Given a finite set of pairwise different labels $l_1, \ldots, l_n$, the session type $\&\{l_1{:}T_1, \ldots, l_n{:}T_n\}$ is given to a channel that *offers* protocols $T_1, \ldots, T_n$. Dually, the type of a channel that *selects* exactly one of those (alternative) behaviors is denoted $\oplus\{l_1{:}T_1, \ldots, l_n{:}T_n\}$. We write $\mathbf{end}$ to denote the completed protocol.

Assuming basic types `item`, `confirmation`, `address`, and `ETA`, the Buyer-Seller-Shipper protocol can be formalized as:

$$BuySell = !\texttt{item}.?\texttt{confirmation}.!\texttt{address}.\mathbf{end}$$
$$SellShip = !\texttt{address}.\mathbf{end}$$
$$ShipBuy = !\texttt{ETA}.\&\{yes : !ok.\mathbf{end}, no : !bye.\mathbf{end}\}$$

Type $BuySell$ describes interactions between Buyer and Seller from Buyer's perspective. Similarly, $SellShip$ describes an interaction between Seller and Shipper from Seller's perspective. $ShipBuy$ takes the standpoint of Shipper in the interaction with Buyer: Shipper first sends a value of type `ETA` to Buyer; then, she/he offers a selection between two labels, denoted yes and no. Each branch outputs a message and then closes the protocol.

An important notion in session-based concurrency is *duality*, which relates session types with opposite behaviors: e.g., the dual of input is output (and vice versa). For example, the dual of $ShipBuy$ above is the session type $BuyShip$:

$$BuyShip = ?\texttt{ETA}.\oplus\{yes :?ok.\mathbf{end}, no :?bye.\mathbf{end}\}.$$

Having described session types, we now move on to motivate a small programming calculus used to specify interacting processes, based on the π-calculus [31]. The following two processes implement session types $ShipBuy$ and $BuyShip$, respectively:

$$shipper = \texttt{send}\ c\ ETA; \texttt{case}\ c\ conf\ \texttt{with}$$

$$\mid \texttt{yes} \rightarrow \texttt{send}\ c\ ok; \texttt{nil}$$
$$\mid \texttt{no} \rightarrow \texttt{send}\ c\ bye; \texttt{nil}$$
$$buyer = \texttt{recv}\ d\ y\ \texttt{in sel}\ d\ \texttt{yes}; \texttt{recv}\ d\ w\ \texttt{in nil}$$

Intuitively, process $shipper$ first sends value ETA (of type ETA) through channel c. Then, using $conf$ as a placeholder for a label, the **case** construct waits for a label along channel c, which will define how successive interactions will proceed: either by confirming the shipping or by rejecting it. We write nil to denote that a channel has been closed. Being based on session type $BuyShip$, process $buyer$ first receives a value channel d replacing bound variable y, and then selects label yes through d. Then it receives a value along d replacing bound variable w, associated to the selected label.

The relationship between processes (such as $shipper$) and session types (such as $ShipBuy$) has been thoroughly studied. It has been shown that session type-checking can enforce important properties of communication, such as protocol compliance and absence of communication errors; advanced systems ensure properties such as progress/deadlock freedom (see, e.g., [14, 37]).

The Need for Events and Timed Behavior in Protocols. While known process languages with session types may adequately describe interactions between processes, they appear to be less suitable to specify other relevant aspects in structured communications, such as timed behavior or contextual information (important to handle, e.g., (partial) failures). In our example, suppose that one of the process implementations (say, $shipper$) fails: we would like to replace the faulty process, and to continue with the established protocol with Buyer and Seller. This failure can be seen as an *event* of the system, very hard to anticipate, and to which other communicating processes should swiftly react. Clearly, this kind of events are relevant for the protocol—failures can jeopardize protocol correctness; still, events cannot be properly captured by the process languages based on the π-calculus, which focus on specifying intended point-to-point synchronizations. While extensions of the session π-calculus with exceptions exist [10, 11], they do not support unanticipated exceptional conditions related to time, events, partial information, nor their combination.

What appears desirable in this case is to complement the protocol specification (lines 1-6 above) with a requirement such as

(I) *If Shipper disconnects from Buyer and Seller then it will eventually reestablish the stipulated protocol.*

This kind of requirements are natural (and sometimes even indispensable) in many practical scenarios (e.g., financial protocols running in mobile devices), and yet they are not easily expressible in frameworks of session-based concurrency that rely on specification languages based on the π-calculus. Although extensions of session-based π-calculi with events and timed behavior exist (see, e.g., [6, 26]), there is not yet a session-based specification language that uniformly captures events, time, and the reactive flavor that the interplay of such aspects unavoidably entails.

As an alternative, we argue that *synchronous reactive programming* (SRP) may offer a natural basis for specifying structured communications, including also rich forms of contextual information and timed behavior that are beyond the scope of languages based on the π-calculus. In particular, this paper proposes ReactiveML as a formal programming calculus for session-based communication.

Our Approach and Contributions. Our interest is in harnessing the key features of SRP in the realm of session-based concurrency. The goal is to provide programmers with a single language in which protocols governed by session types can be written and verified, incorporating requirements such as (I) in protocol implementations. As a first step, here we develop a correct implementation of the essential constructs of session-based concurrency in ReactiveML. This entails relating two formal frameworks: on the one hand, we have a process language based on the π-calculus (here called CPL, based on the model language in [37]); on the other hand, we have ReactiveML, whose formal foundations are given in [28, 29].

A main obstacle to the desired relation is that ReactiveML lacks an explicit notion of *channels*, which is essential in well-established execution models of session-based concurrency.

In ReactiveML, synchronization occurs via *signals* (i.e., events) that allow processes to have a deterministic reaction to input events. In ReactiveML, an *instant* is a logical time unit whose duration is given by the emission of signals in a program. Indeed, a process may react to input events by emitting new signals, which in turn can keep affecting the internal execution of the program during that instant. Once all possible emissions have been made, the program completes the current instant, and tests if the absence of signals triggers new behaviors, to be executed in the next instant. That is, when an instant ends no further signals can be emitted.

We briefly discuss key constructs in ReactiveML. Given a signal s and a value v, we have the expressions:

$$\mathsf{emit}\ s\ v \qquad \mathsf{await}\ s(\tilde{x})\ \mathsf{in}\ P$$

While $\mathsf{emit}\ s\ v$ specifies that value v can be sent along signal s on the current instant, $\mathsf{await}\ s(\tilde{x})\ \mathsf{in}\ P$ is used to "listen" on signal s: a synchronization only occurs when a signal is caught (with placeholders $\tilde{x}$), and the continuation P will only execute in the next instant. Two other constructs useful in our work are pause, which leaves the process in a suspended state until the next instant, and $\mathsf{signal}\ c\ \mathsf{in}\ P$, which is a simplified variant of the construct that declares a new (fresh) signal c within the scope of P.

As mentioned above, there are conceptual differences between the models in which session-based concurrency has

been usually investigated, and the computation model of ReactiveML. To address these differences, we propose a "signals-as-channels" approach to model the communication primitives of session-based concurrency in ReactiveML. Roughly speaking, this means using the capability of emitting valued signals in ReactiveML to simulate the use of channels in a communication-based setting.

Another technical difficulty concerns *polymorphism*. In session-based concurrency, channels are assumed polymorphic, for they are expected to carry over different types of values during the execution of a protocol. However, signals in ReactiveML are not polymorphic: the type of data that signals can carry is defined from the beginning and cannot be changed during execution.

To address this issue, and borrowing inspiration from [15], our implementation of sessions in ReactiveML follows a *continuation-passing style*. Intuitively, the idea is that every signal that carries a message of a given value is used exactly once; the structure of the protocol is passed around in the continuation. This kind of translation eliminates the need for polymorphic signals.

We define our implementation as a formal translation (interpretation) $[\![\cdot]\!]$ that gives a ReactiveML program for each construct in CPL, a session-based π-calculus. Thus, our translation sheds light on how to enhance ReactiveML with practical constructs for communication-centric programming. This way, e.g., primitives for output and input in CPL are modeled in ReactiveML as follows:

$$[\![\texttt{send}\ c\ v; P]\!] = \mathsf{signal}\ c'\ \mathsf{in}\ \mathsf{emit}\ c\ (v, c'); \mathsf{pause}\,; [\![P]\!]$$
$$[\![\texttt{recv}\ c\ y\ \texttt{in}\ P]\!] = \mathsf{await}\ c(y, w)\ \mathsf{in}\ [\![P]\!]$$

where we assume that w is a fresh name in P.

A key design choice in our interpretation is to represent sequential actions (decreed by session types) by enforcing that each emission/awaiting synchronization to occur in a single instant. As a result, each sequential step of the protocol lasts an instant. This choice is intended to preserve the sequential structure of communication protocols. Hence, as shown above, each signal emission will be then followed by a pause construct. Note that in ReactiveML an end-of-instant occurs when no other signal can be emitted.

3. Preliminaries

We now formally introduce the two models that we will relate: session-based concurrency, in the context of a simple programming language with communication primitives, based on the π-calculus introduced in [37], and ReactiveML, as introduced in [28, 29].

3.1 CPL: A Core Language with Communication

CPL is a core programming language, a variant of the (synchronous) session π-calculus [23] with an intuitive syntax.[1]

Syntax. CPL relies on a countable infinite set of variables $\mathcal{V}$, ranged over by $x, y, \ldots$. We use $v, v', \ldots$ to range over variables and constants (*values*). Also, we use $l, l', \ldots$ to range over *labels*. We consider also *channels*, a class of variables; to distinguish them from variables, we use $c, c', \ldots, d, d' \ldots$ range over channels.

$$\begin{aligned} P, Q ::= {} & \texttt{send}\ x\ v; P \mid \texttt{recv}\ x\ y\ \texttt{in}\ P \mid \texttt{sel}\ x\ l; P \\ & \mid \texttt{case}\ x\ l\ \texttt{with}\ \{l_i \to P_i\}_{i\in I} \mid \texttt{nil} \mid P \mid Q \\ & \mid \texttt{rec}\ X.P \mid \texttt{new}\ x, y\ \texttt{in}\ P \mid X \end{aligned}$$

Process $\texttt{send}\ x\ v; P$ sends value v over channel x and continues as P. Process $\texttt{recv}\ x\ y\ \texttt{in}\ P$ receives a value v over channel x and then continues as $P\{^v/_y\}$, i.e., process P in which all free occurrences of y are replaced with v. The branching operator $\texttt{case}\ x\ l\ \texttt{with}\ \{l_1 \to P_1, \cdots, l_n \to P_n\}_{i\in I}$ offers alternatives $P_1, \ldots, P_n$, identified by pairwise distinct labels $l_1, \ldots, l_n$. Branching is meant to interact with the selection operator $\texttt{sel}\ x\ l; P$, which selects one of the labeled alternatives. Process $\texttt{new}\ x, y\ \texttt{in}\ P$ is (almost) the $(\nu xy)P$ π-calculus restriction: it declares *(session) endpoints* x and y private to P. Process $P \mid Q$ denotes the parallel (interleaved) execution of P and Q. To express recursive behavior, we define X as a process variable that can be used inside process P in the recursive construct $\texttt{rec}\ X.P$. The set of free variables of a process P, denoted $fv(P)$, is defined as usual; we assume processes $\texttt{recv}\ x\ y\ \texttt{in}\ P$ and $\texttt{new}\ x, y\ \texttt{in}\ P$ bind variables y and x, y respectively, with scope P.

Operational Semantics. The semantics for CPL processes is given as a reduction relation, denoted $\to_c$, defined as the smallest relation generated by the rules in Fig. 1. Reduction expresses the computation steps that a process performs on its own. It relies on a structural congruence relation, denoted $\equiv$, which identifies processes up to consistent renaming of bound names, denoted $\equiv_\alpha$. Formally, $\equiv$ is the smallest congruence that satisfies the axioms:

$$P \mid \texttt{nil} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \equiv Q \text{ if } P \equiv_\alpha Q$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad \texttt{new}\ c\ \texttt{in}\ \texttt{nil} \equiv \texttt{nil}$$
$$\texttt{new}\ c, c'\ \texttt{in}\ \texttt{new}\ d, d'\ \texttt{in}\ P \equiv \texttt{new}\ d, d'\ \texttt{in}\ \texttt{new}\ c, c'\ \texttt{in}\ P$$
$$(\texttt{new}\ c, d\ \texttt{in}\ P) \mid Q \equiv \texttt{new}\ c, d\ \texttt{in}\ P \mid Q \text{ if } c, d \notin fv(Q).$$

We give some intuitions on the reduction rules. Following [37], we define reductions as synchronizations that occur in the scope of the restriction operator. This is because session communication should occur between endpoints without any external interference. Hence, in $\texttt{new}\ x, y\ \texttt{in}\ P$ restriction plays two roles: binding variables and defining dual

[1] A note on terminology: *synchronous communication* as in the π-calculus should not be confused with *synchronous programming* as in ReactiveML. The distinction between synchronous and asynchronous communication in the π-calculus typically relies on assumptions on output actions and auxiliary devices for messages (e.g., queues or buffers). In the *synchronous* π-calculus, output actions are blocking and there are no queues/buffers.

$$\lfloor \textsc{Comm} \rfloor \ \texttt{new}\ c, d\ \texttt{in}\ \texttt{send}\ c\ v; P \mid \texttt{recv}\ d\ y\ \texttt{in}\ Q \rightarrow_c \texttt{new}\ c, d\ \texttt{in}\ P \mid Q\{v/y\}$$

$$\lfloor \textsc{Sel} \rfloor \ \texttt{new}\ c, d\ \texttt{in}\ \texttt{sel}\ c\ l_j; P \mid \texttt{case}\ c\ l\ \texttt{with}\ \{l_i \rightarrow Q_i\}_{i \in I} \rightarrow_c \texttt{new}\ c, d\ \texttt{in}\ P \mid Q_j \text{ if } j \in I$$

$$\lfloor \textsc{Rec} \rfloor \ \texttt{rec}\ X.P \rightarrow_c P\{\texttt{rec}\ X.P/X\}$$

$$\lfloor \textsc{Str} \rfloor \ P \equiv P',\ P' \rightarrow_c Q',\ Q' \equiv Q \Rightarrow P \rightarrow_c Q$$

Figure 1. Reduction rules for CPL.

endpoints. Rule $\lfloor$COMM$\rfloor$ enables input/output synchronization. Rule $\lfloor$SEL$\rfloor$ formalizes labeled, deterministic choice, resulting from the synchronization of selection and branching operators. Rule $\lfloor$REC$\rfloor$ defines recursion unfolding in a customary manner. Rule $\lfloor$STR$\rfloor$ allows us to use structural congruence within reductions.

Session Types for CPL. We summarize the type system for CPL, which is based on the type system for the π-calculus in [37]. Complete details are given in [3].

Session types are defined from the following syntax:

$$T ::= B \mid ?T.T \mid !T.T \mid \oplus\{l_i : T_i\}_{i \in I} \mid \&\{l_i : T_i\}_{i \in I} \mid \mathbf{end}$$

In [37] these syntactic structures are called *pretypes*. Pretype B is assumed to be any basic type (e.g., booleans). Pretype $?T_1.T_2$ denotes input, and types a channel that sends a value of type T_1 and continues according to type T_2. Dually, pretype $!T_1.T_2$ denotes output, and types a channel that receives a value of type T_1 and then proceeds according to type T_2. Pretypes $\oplus\{l_i : T_i\}_{i \in I}$ and $\&\{l_i : T_i\}_{i \in I}$ denote labeled selection (internal choice) and branching (external choice), respectively. Pretype $\mathbf{end}$ is the type of a channel that can no longer be used.

In [37], a *type* is defined as a prefixed pretype. Prefixes for pretypes can be lin or un. A lin prefix indicates a session whose endpoint can only be used in one *thread* (i.e., a process not comprising parallel composition). A type prefixed by un indicates that the endpoint can be shared between multiple threads.

Type checking analyzes processes with respect to a *context* Γ. Contexts assign types to endpoints. Typing judgments are then of the form $\Gamma \vdash P$; the typing rules can be found in [3].

Our operational correspondence results rely on auxiliary definitions associated to the structure of processes and their typability:

Definition 3.1 (Single-Session Processes). *A process P is* single-session *if it is sequential (i.e., it does not contain parallel sub-processes) and contains exactly one session endpoint.*

Definition 3.2 (Programs). *A process P without free variables is called* a program. *A program P is typable if it is well-typed under the empty context.*

We will focus on *single-session programs*, well-typed programs that result from composing two single-session processes. Process

$$\texttt{new}\ x, y\ \texttt{in}\ (\texttt{send}\ x\ v_1; \texttt{recv}\ x\ w\ \texttt{in}\ \texttt{nil} \mid \texttt{recv}\ y\ z\ \texttt{in}\ \texttt{send}\ y\ v_2; \texttt{nil})$$

is a single-session typable program. In contrast, the process:

$$\texttt{new}\ x_1, y_1\ \texttt{in}\ \texttt{new}\ x_2, y_2\ \texttt{in}\ (\texttt{send}\ x_1\ v_1; \texttt{recv}\ x_2\ w\ \texttt{in}\ \texttt{nil} \mid \texttt{recv}\ y_1\ z\ \texttt{in}\ \texttt{send}\ y_2\ v_2; \texttt{nil})$$

is not, since each parallel component contains more than one endpoint (x_1, x_2 and y_1, y_2, respectively).

3.2 ReactiveML

ReactiveML [29] is an extension of OCaml based on the reactive model given in [8], which allows unbounded time response from processes and avoids causality issues that can occur in other approaches to SRP, such as the one used by ESTEREL [5]. ReactiveML extends OCaml with *processes*, which are state machines whose behavior can be executed through several logical instants. Processes are considered the reactive counterpart of OCaml functions, which are executed instantaneously in ReactiveML.

In ReactiveML, synchronization is based on *signals*: events that occur in one logical instant. Signals can trigger reactions in processes. In turn, these reactions can be executed instantaneously or in the next instant. Signals can carry values and can be emitted from different processes in the same logical instant.

We present the syntax and semantics of ReactiveML as given in [28], where a type-and-effect system is defined to differentiate *cooperative* from *non-cooperative* programs. Intuitively, a well-typed cooperative program in [28] will not contain so-called *instantaneous loops* which may not give control back to the scheduler.

Syntax. Let v denote values and e denote expressions as in an ML-like language. Then we have:

$$v ::= c \mid (v, v) \mid n \mid \lambda x.e \mid \mathsf{process}\ e$$

$$e ::= x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \mathsf{rec}\ x = v \mid \mathsf{run}\ e \mid \mathsf{pause} \mid \mathsf{process}\ e \mid \mathsf{emit}\ e\ e \mid e?e : e \mid \mathsf{loop}\ e \mid \mathsf{let}\ x = e\ \mathsf{and}\ x = e\ \mathsf{in}\ e \mid \mathsf{signal}_e\ x : e\ \mathsf{in}\ e \mid \mathsf{do}\ e\ \mathsf{when}\ e \mid \mathsf{do}\ e\ \mathsf{until}\ e(x) \rightarrow e \mid \mathsf{match}\ c\ \mathsf{with}\ \{c_i \rightarrow e_i\}_{i \in I}$$

The syntax builds upon standard OCaml constructs. Below we will use _ as a placeholder for any value and () to denote

$$e_1 \parallel e_2 \triangleq \mathsf{let}\ \_ = e_1\ \mathsf{and}\ \_ = e_2\ \mathsf{in}\ ()$$
$$e_1; e_2 \triangleq \mathsf{let}\ \_ = ()\ \mathsf{and}\ \_ = e_1\ \mathsf{in}\ e_2$$
$$\mathsf{await}\ e_1(x)\ \mathsf{in}\ e_2 \triangleq \mathsf{do\ loop\ pause\ until}\ e_1(x) \to e_2$$

Figure 2. Derived constructs for ReactiveML.

the unit process. The syntax of values includes the constant value c, the pair of values (v, v), λ-functions, as well as process declarations.

Processes are made of expressions. Expression x denotes a variable and c denotes a constant; (e, e) denotes a pair of expressions. Functions and applications are also expressions. Expression $\mathsf{rec}\ x = v$ denotes a recursive process. Process $\mathsf{run}\ p$ indicates that p will be executed. Expression pause denotes that a process will not execute its continuation until the next instant. The let construct is defined as in ML-like languages. The signal declaration $\mathsf{signal}_g\ x\ :\ t\ \mathsf{in}\ e$ declares a signal x with type t, which will be bound in the continuation e; in this declaration, g is a *gathering function*, which gathers all the values of the signal in one instant. The construct $\mathsf{emit}\ e_1\ e_2$ emits a (possibly valued) signal e_2 along e_1. The presence evaluation construct $s?p : q$ tests the presence of signal s: if s is present then p is executed in the same instant; otherwise, q is executed in the next instant. The $\mathsf{loop}\ e$ construct executes e in an infinite loop. The construct $\mathsf{do}\ p\ \mathsf{when}\ s$ executes p only when signal s is present, and suspends its execution otherwise. The construct $\mathsf{do}\ p\ \mathsf{until}\ s(x) \to q$ executes p until a signal s with value x is emitted. If this occurs, the execution of p stops at the end of the instant and q is executed in the next instant. Besides these basic constructs, we will use the derived constructs in Fig. 2.

Semantics of ReactiveML programs. We present the big-step semantics of ReactiveML, given as a *Labeled Transition System* (LTS). We prefer the big-step semantics over the small-step semantics in [29], because in the LTS each transition corresponds to one logical instant, which is convenient in our operational correspondence result. Both small- and big-step semantics coincide [3].

The declared signals are stored in a *signal environment* S. This environment assigns triplets containing all the information related to a signal (default value, gathering function and a multiset of emitted values) to a single signal name. We denote $S^d(n_i)$ as the default value of signal n_i, $S^g(n_i)$ as the gathering function of n_i and $s^m(n_i)$ as the multiset value of n_i. We will denote S^m as a function that maps every signal name in S to its corresponding multiset. We also have *events*, denoted $E, E', \ldots$, which contain all the information of the output signals emitted by a program: they map signal names to a multiset of emitted values. Events can change throughout a program's execution. Events come with a union operator and an inclusion relation (denoted $\sqcup_E$ and $\sqsubseteq_E$), which are defined pointwise in each multiset contained in the event.

Notation 3.3 (Multiset). When giving an extensional description of a multiset, we will use the notation $\wr e_1, \ldots, e_n \int$.

A big-step transition in ReactiveML captures reactions within a single instant. At each instant i, the program reads an input I_i and produces an output O_i. The reaction of an expression is then defined by the smallest signal environment S_i (wrt $\sqsubseteq_S$) such that:

$$e_i \xrightarrow[S_i]{E_i, b_i} e_{i+1}$$

where: 1) $(I_i \sqcup_E E_i) \sqsubseteq_E S_i^m$; 2) $O_i \sqsubseteq_E E_i$; 3) $S_i^d \subseteq S_{i+1}^d$; and 4) $S_i^g \subseteq S_{i+1}^g$. These conditions are explained as follows:

(1) S must contain the inputs I_i and emitted signals E_i.
(2) The output signals O_i are included in the emitted signals E_i.
(3) Default values are preserved from one instant to another.
(4) Gathering functions are preserved from one instant to another.

For space reasons, Fig. 3 shows only a few LTS rules; see [3] for details. Some intuitions follow. Rule $\lfloor$CASE$\rfloor$ checks that there is a match between constant c_j and some of the constants c_i (with $i \in I$). Rule $\lfloor$EMIT$\rfloor$ ensures that the new value is added to the signal n that emitted it. Rule $\lfloor$DOU-END$\rfloor$ concerns the termination of a do/until construct, when the internal expression e_1 reaches value v. Rule $\lfloor$DOU-PRE$\rfloor$ says that a do/until expression executes its continuation in the next instant, given that signal n is in S. Rule $\lfloor$DOU-NPRE$\rfloor$ says that the internal body of a do/until expression can execute over several instants. Rule $\lfloor$PAUSE$\rfloor$ uses the termination boolean b: if $b = \mathtt{tt}$ (true) then the expression terminates in that instant. Otherwise, if $b = \mathtt{ff}$ (false), the process is suspended (stuck) until the next instant.

To distinguish between usual OCaml expressions and ReactiveML expressions, we use a *well-formedness* predicate, denoted $k \vdash e$ for some expression e. If $k = 0$ then e is an instantaneous (usual ML) expression; otherwise, if $k = 1$, then e is a reactive expression. See [3] for details.

Properties. We state two main results for the big-step semantics of ReactiveML, as in [29]. Details for these theorems are in [3].

First, we prove that the semantics is deterministic, assuming that the gathering functions are associative and commutative.

Theorem 3.4 (ReactiveML is deterministic). *Let e be an arbitrary well-formed ReactiveML expression. Also, assume an arbitrary signal environment S, arbitrary events E_1, E_2, and arbitrary termination values b_1, b_2. Suppose (1) $e \xrightarrow[S]{E_1, b_1} e_1$, (2) $e \xrightarrow[S]{E_2, b_2} e_2$ and (3) $\forall n \in Dom(S)$,*

$$\lfloor \textsc{Case} \rfloor \frac{j \in I \qquad e_j \xrightarrow[S]{E,b} e'_j}{\mathsf{match}\ c_j\ \mathsf{with}\ \{c_i \to e_i\}_{i\in I} \xrightarrow[S]{E,b} e'_j} \qquad \lfloor \textsc{Emit} \rfloor \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} n \qquad e_2 \xrightarrow[S]{E_2,\mathtt{tt}} v}{\mathsf{emit}\ e_1\ e_2 \xrightarrow[S]{E_1 \sqcup_E E_2 \sqcup_E [\{v\}/n],\mathtt{tt}} ()}$$

$$\lfloor \textsc{DoU-End} \rfloor \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \qquad e_1 \xrightarrow[S]{E_1,\mathtt{tt}} v}{\mathsf{do}\ e_1\ \mathsf{until}\ e_2(x) \to e_3 \xrightarrow[S]{E_1 \sqcup_E E_2,\mathtt{tt}} v} \qquad \lfloor \textsc{DoU-Pre} \rfloor \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \qquad n \in S \qquad e_1 \xrightarrow[S]{E_1,\mathtt{ff}} e'_1}{\mathsf{do}\ e_1\ \mathsf{until}\ e_2(x) \to e_3 \xrightarrow[S]{E_1 \sqcup_E E_2,\mathtt{ff}} e_3\{S^v(n)/x\}}$$

$$\lfloor \textsc{DoU-NPre} \rfloor \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \qquad n \notin S \qquad e_1 \xrightarrow[S]{E_1,\mathtt{ff}} e'_1}{\mathsf{do}\ e_1\ \mathsf{until}\ e_2(x) \to e_3 \xrightarrow[S]{E_1 \sqcup_E E_2,\mathtt{ff}} \mathsf{do}\ e'_1\ \mathsf{until}\ e_2(x) \to e_3} \qquad \lfloor \textsc{Pause} \rfloor \frac{}{\mathsf{pause} \xrightarrow[S]{\emptyset,\mathtt{ff}} ()}$$

Figure 3. Transition for ReactiveML expressions (selected rules).

$S^g(n) = f$, where f is associative and commutative. Then $E_1 = E_2, b_1 = b_2$ and $e_1 = e_2$.

The following result says that there exists a unique, smallest signal environment in which a given expression can react. It ensures that there are no causality paradoxes in a program's execution (e.g., a signal that is present and absent in the same time instant).

Theorem 3.5. *For every expression e, let*

$$\mathcal{S} = \{S \mid \exists E, b : e \xrightarrow[S]{E,b} e'\}$$

Then there exists a (unique) smallest signal environment ($\sqcap\mathcal{S}$) such that $e \xrightarrow[\sqcap\mathcal{S}]{E,b} e'$.

4. Interpreting Sessions in ReactiveML

We present our interpretation of session primitives in ReactiveML. We prove that it satisfies a basic form of operational correspondence, which bears witness of its consistency.

4.1 Interpretation

Our interpretation exploits valued signals in ReactiveML: we use them as carriers for messages, mimicking the sending of messages via channels, as typical in traditional models for communicating systems. This emission is then listened (i.e., awaited) by the communication partner, simulating a receiving action.

As explaied above, a key design choice is the use of a continuation-passing style interpretation to deal with the following issues:

1. Signals are not polymorphic: in ReactiveML, signals are assigned a default, unmodifiable value that defines their type.
2. Linearity of resources in session-based communication: to avoid mismatches, session endpoints should be used exactly once; this is enforced by the session type system.

Our use of the continuation-passing interpretation follows [15]: each time a signal is emitted, we create a new signal. Then the value of emission carries both the initial message and a reference to the new signal, where further synchronizations will occur.

Definition 4.1 (Interpretation). *Let $[\![\cdot]\!]_f$ be a function from CPL processes to ReactiveML defined as in Fig. 4, where f is a function that goes from variables in CPL to variables in ReactiveML.*

The f in $[\![\cdot]\!]_f$ denotes a function that keeps track of name substitutions during the translation of CPL processes. We write f_x as a shorthand for $f(x)$ (i.e., f applied to x).

Some intuitions on the interpretation of Fig. 4 follow: output and selection operators are defined as signal emissions with a continuation that will be executed in the next instant. Considering an interpretation using a continuation-passing style, the emission done in both output and selection operators sends a pair with value v (label l) and a reference to the newly created signal x'. After the emission, the output/selection translations wait for the next instant to execute the continuation $[\![P]\!]_f$. Translations for input and branching wait for the emission of a signal; its value is a pair that contains the expected value and a reference to a new signal where further synchronizations will occur. The parallel composition operator is mapped homomorphically. The `nil` operator is mapped to $()$. Lastly, the translation for restriction in CPL is a process that creates a fresh signal and does the corresponding name substitutions.

4.2 Operational Correspondence

Following [20], we define operational correspondence as a formal property of interpretations that ensures its correctness in terms of executed behaviors. Considering CPL and ReactiveML as *source* and *target* languages, respectively, operational correspondence is divided into *soundness* and *completeness* properties: the former intuitively says that each step in the source language can be matched in the target language using the interpretation; the latter ensures the converse guarantee. We will need an auxiliary definition:

Definition 4.2 (Equivalence of ReactiveML Processes). *We will denote $\equiv_r$ to an equivalence relation given by the*

$$
\begin{aligned}
[\![\texttt{send}\ x\ v; P]\!]_f &\triangleq \mathsf{signal}_g\ x' : (\_,\_)\ \mathsf{in}\ \mathsf{emit}\ f_x\ (v, x'); \mathsf{pause}\,; [\![P]\!]_{f,\{x\leftarrow x'\}} \\
[\![\texttt{recv}\ x\ y\ \texttt{in}\ P]\!]_f &\triangleq \mathsf{await}\ x(y,w)\ \mathsf{in}\ [\![P]\!]_{f,\{x\leftarrow w\}} \\
[\![\texttt{sel}\ x\ v; P]\!]_f &\triangleq \mathsf{signal}_g\ x' : (\_,\_)\ \mathsf{in}\ \mathsf{emit}\ f_x\ (l, x'); \mathsf{pause}\,; [\![P]\!]_{f,\{x\leftarrow x'\}} \\
[\![\texttt{case}\ x\ l_j\ \texttt{with}\ \{l_i \to P_i\}_{i\in I}]\!]_f &\triangleq \mathsf{await}\ x(y,w)\ \mathsf{in}\ \mathsf{match}\ l\ \mathsf{with}\ |\, l_1 \to [\![P_1]\!]_{f,\{x\leftarrow w\}} \cdots |\, l_n \to [\![P_n]\!]_{f,\{x\leftarrow w\}} \\
[\![P \mid Q]\!]_f &\triangleq [\![P]\!]_f \mid [\![Q]\!]_f \\
[\![\texttt{new}\ x, y\ \texttt{in}\ P]\!]_f &\triangleq \mathsf{signal}_g\ c : (\_,\_)\ \mathsf{in}\ [\![P]\!]_{f,\{x\leftarrow c, y\leftarrow c\}} \\
[\![\texttt{rec}\ X.P]\!]_f &\triangleq \mathsf{rec}\ X = [\![P]\!]_f \\
[\![\texttt{nil}]\!]_f &\triangleq ()
\end{aligned}
$$

Figure 4. Translation from CPL to ReactiveML.

following rule: match c_j with $\{c_i \to P_i\}_{i\in I} \equiv_r P_j$ *if* c_j *is a constant and* $j \in I$ *and the identity equivalence for every other expression.*

We then have:

Theorem 4.3 (Operational Correspondence). *Let* P, Q *be CPL single-session well-typed programs. Also, let* f *be a renaming function and let* S, E *be a signal environment and an event, respectively. Then:*

1. *(soundness) If* $P \to Q$*, then there exist* R, S, E *such that* $[\![P]\!]_f \xrightarrow[S]{E,\texttt{ff}} R$ *and* $R \equiv_r [\![Q]\!]_f$.
2. *(completeness) If* $[\![P]\!]_f \xrightarrow[S]{E,\texttt{ff}} R$ *then there exists* Q *such that* $P \to Q$ *and* $[\![Q]\!]_f \equiv_r R$.

This result ensures the correctness and faithfulness of our formal implementation of session-based communicating processes. Our design choices give us two main guarantees: (1) Thanks to the big-step semantics of ReactiveML, every step in CPL is observed as occurring in a single time unit, allowing us to have a notion of sequence of the communication actions, and (2) Polymorphism of channels is preserved via continuation-passing style, allowing for a transparent translation between CPL and ReactiveML.

5. Revisiting the Buyer-Seller-Shipper

Translation. We translate processes $shipper$ and $buyer$ from § 2. Below, c, d are dual endpoints.

$[\![shipper]\!]_{\{c\leftarrow e\}} =$

$\quad \mathsf{signal}_g\ c' : (\_,\_)\ \mathsf{in}\ \mathsf{emit}\ e\ (ETA, c'); \mathsf{pause}\,;$

$\quad \mathsf{await}\ c'(y,w)\ \mathsf{in}$

$\quad \mathsf{match}\ y\ \mathsf{with}$

$\quad\quad |\ \mathtt{yes} \to \mathsf{signal}_g\ c'' : (\_,\_)\ \mathsf{in}\ \mathsf{emit}\ w\ (ok, c''); \mathsf{pause}$

$\quad\quad |\ \mathtt{no} \to \mathsf{signal}_g\ c'' : (\_,\_)\ \mathsf{in}\ \mathsf{emit}\ w\ (bye, c''); \mathsf{pause}$

$[\![buyer]\!]_{\{d\leftarrow e\}}$

$\quad \mathsf{await}\ e(y,w)\ \mathsf{in}\ \mathsf{signal}_g\ d : (\_,\_)\ \mathsf{in}\ \mathsf{emit}\ w\ (yes, d);$

$\quad \mathsf{pause}\,; \mathsf{await}\ d(y,w)\ \mathsf{in}\ ().$

By using the reduction semantics of CPL, we observe that process $\texttt{new}\ c, d\ \texttt{in}\ (shipper \mid buyer) \to_c^3 \texttt{nil}$. To give intuitions on the operational correspondence statement, we show that by using the big-step semantics of ReactiveML on

$$R_0 = ([\![shipper]\!]_{\{c\leftarrow e\}} \mid [\![buyer]\!]_{\{d\leftarrow e\}})$$

we may mimic these steps and reach process $()$ in three transitions. First, consider the following signal environment and events:

$$
\begin{aligned}
S &= \{((ETA, c'), g, \wr(ETA, c')\int)/c, ((yes, d), g, \wr(yes, d)\int)/c', \\
&\qquad ((yes, c''), g, \wr(ok, c'')\int)/d, ((\_,\_), g, \wr\int)/c''\} \\
E_1 &= \{\wr(ETA, c')\int/c\} \\
E_2 &= \{\wr(ETA, c')\int/c, \wr(yes, d)\int/c'\} \\
E_3 &= \{\wr(ETA, c')\int/c, \wr(yes, d)\int/c', \wr(ok, c'')\int/d\}.
\end{aligned}
$$

Then, the transition proceeds as follows:

$$R_0 \xrightarrow[S]{E_1,\texttt{ff}} (R_1 \mid R_2) \xrightarrow[S]{E_2,\texttt{ff}} (R_1' \mid R_2') \xrightarrow[S]{E_3,\texttt{ff}} ()$$

where R_1, R_2, R_1', and R_2' are as in Fig. 5. Each transition determines a time instant. This agrees with our design choices, in which each communication action is done in one logical instant. This allows us to give a timed interpretation of session protocols in CPL.

A Reactive Buyer-Seller-Shipper Implementation. Our objective is to endow ReactiveML with constructs for session-based concurrency, so to have a uniform basis for communication-centric programs with timed and event-based behavior. To highlight the advantages of specifying sessions in a synchronous reactive setting, we present a simple reactive extension to the previous example.

Assume two implementations for Shipper: one has a faulty connection with Buyer ($shipper1$); the other is a

$R_1 = \mathsf{await}\ c'(y,w)\ \mathsf{in}\ \ \mathsf{match}\ y\ \mathsf{with}$

$\quad \mid \mathtt{yes} \rightarrow \mathsf{signal}_g\ c'' : (\_,\_)\ \mathsf{in\ emit}\ w\ (ok, c''); \mathsf{pause}$

$\quad \mid \mathtt{no} \rightarrow \mathsf{signal}_g\ c'' : (\_,\_)\ \mathsf{in\ emit}\ w\ (bye, c''); \mathsf{pause}$

$R_2 = \mathsf{signal}_g\ d : (\_,\_)\ \mathsf{in\ emit}\ c'\ (yes, d); \mathsf{pause}\,;$

$\quad \mathsf{await}\ d(y,w)\ \mathsf{in}\ ()$

$R_1' = \mathsf{match}\ yes\ \mathsf{with}$

$\quad \mid \mathtt{yes} \rightarrow \mathsf{signal}_g\ c'' : (\_,\_)\ \mathsf{in\ emit}\ d\ (ok, c''); \mathsf{pause}$

$\quad \mid \mathtt{no} \rightarrow \mathsf{signal}_g\ c'' : (\_,\_)\ \mathsf{in\ emit}\ d\ (bye, c''); \mathsf{pause}$

$R_2' = \mathsf{await}\ d(y,w)\ \mathsf{in}\ ().$

Figure 5. The Buyer-Seller-Shipper in ReactiveML (cf. § 5)

backup ($shipper2$). A connection error can occur at any moment. We simulate this with signal k that *kills* the faulty $shipper1$. Upon disconnection, $shipper1$ sends a non-valued signal to $shipper2$; the backup code will then re-establish the connection with Buyer. Assume that $shipper\ e$ and $buyer\ c$ are the processes that result from the translations $[\![shipper]\!]_{\{c \leftarrow e\}}$ and $[\![buyer]\!]_{\{d \leftarrow e\}}$ respectively, and the parameter e corresponds to signal that will be substituted in the encoded processes. We have the following ReactiveML processes:

$\mathsf{let\ process}\ shipper1\ c\ k\ d =$

$\quad \mathsf{do}\ shipper\ c\ \mathsf{until}\ k \rightarrow \mathsf{emit}\ d\ ()$

$\mathsf{let\ process}\ buyerR\ c\ d\ r =$

$\quad \mathsf{do}\ buyer\ c\ \mathsf{until}\ d \rightarrow \mathsf{await}\ r(c')\ \mathsf{in}\ buyer\ c'$

$\mathsf{let\ process}\ shipper2\ c\ d\ r =$

$\quad \mathsf{await}\ d\ \mathsf{in}\ \mathsf{signal}_g\ c' : \_\ \mathsf{in\ emit}\ r\ c'; \mathsf{pause}\,; shipper\ c'.$

Above, there are four global signals: c, d, r, k. Process $shipper1$ synchronizes on signal c with $buyerR$. The emission of the (kill) signal k indicates an error. At this point, $shipper1$ emits a non-valued signal d, which is awaited by $shipper2$. Subsequently, $shipper2$ and $buyerR$ can re-establish the session: a signal r is emitted by $shipper2$ and received by $buyerR$. Signal r has a reference to signal c' where further synchronizations will occur.

Note that the reactive interaction pattern in the above example cannot be expressed in CPL, because it lacks constructs to model changes in the flow of the program due to interruptions. While session π-calculi extended with exceptions exist [11], these extensions tend to have a contrived underlying theory, and to focus on one particular phenomenon, neglecting other (useful) features. For example, the work in [11] does not consider timed, reactive behavior, which is natural to ReactiveML.

6. Concluding Remarks and Future Work

We have reported ongoing work on the first interpretation of session-based concurrency in synchronous reactive programming. Relying on signals as main synchronization mechanism, our interpretation defines "blocks" of communication-centric code, which can be embedded in larger ReactiveML programs; it is supported by an operational correspondence result (Thm. 4.3). By means of examples, we have shown that the interpretation enables natural ways of expressing timed and event-based behavior in session-based programs, which exploit constructs for explicit preemption and suspension, typical of SRP models. We focused on ReactiveML as a practical programming framework in which our interpretation can be implemented and made available to protocol programmers.

Future Work. We plan to extend our development to account for asynchronous (queue-based) communication in SRP. Since signal emission in ReactiveML is asynchronous, an interpretation into a CPL with explicit queues (for collecting messages from the valued signals) should be more natural than the approach developed here.

As mentioned above, ReactiveML comes with a type-and-effect system for checking *cooperativity* [28]. Informally, such a system statically rules out improductive/infinite reduction sequences. We plan to study whether cooperativity is related to correctness guarantees in session-based concurrency, such as deadlock-freedom. This could lead to alternative verification mechanisms for programs.

7. Related Work

SRP was introduced in the 1980s [4] as a way to implement and design critical real-time systems. Since then, several works have provided solid foundations for SRP programming languages. In particular, the work on ESTEREL [5] and the model presented in [8] offer foundations for languages such as ReactiveML [28, 29] and ULM [7]. Also worth mentioning are works that relate synchronous languages to the π-calculus; for instance, the work [1] develops a non-deterministic variant of the SRP model of ESTEREL. The paper [21] offers a survey of synchronous reactive programming languages, including ESTEREL, LUSTRE [12], and several others.

Session types [23] have been thoroughly studied. Prior works have extended the foundations of session-based concurrency to include event-based behavior [26], adaptive behavior [13], and timed behavior [6]. All these extensions use (variants of) the π-calculus as their base language. A key difference with our work is that we propose an SRP language (i.e., ReactiveML) to obtain a natural integration of some of the aforementioned features. Practical approaches to session types have resulted in a variety of implementations, including [32, 35, 38]. The paper [2] offers a recent survey of session types and behavioral types in practice.

Another relevant implementation is [34], a source of inspiration for our work: it integrates session-based concurrency in the OCaml programming language. As in our interpretation, the implementation in [34] uses the notion of continuation-passing style developed in [15]. A distinguishing feature of our work with respect to [34] is our interest in reactive, timed behaviors, not supported by OCaml, and therefore not available in [34]. Our current implementation still lacks some features present in [34], such as the integration of duality and linearity-related checks into the OCaml type system.

Our approach is related to our prior works on declarative interpretations of session π-calculi [9, 27]. The first such interpretation is developed in [27], where it is shown that declarative languages can support *mobility* in the sense of the π-calculus. The interpretation developed in [9] improves over [27] by supporting linearity and non-determinism. The works [9, 27] are related to the present work due to the declarative flavor of SRP. In contrast, our reactive interpretation yields practical implementations in ReactiveML, which are not possible in the foundational interpretations in [9, 27].

Outside process algebraic formalisms (and type-based validation techniques), other approaches to the formal specification and analysis of services use automata- and graph-based techniques. For instance, the work [18] uses Büchi automata to specify and analyze the conversation protocols that underlie electronic services.

Acknowledgments

We are grateful to the anonymous reviewers for their useful suggestions. This research was partially supported by EU COST Action IC1201 (Behavioral Types for Reliable Large-Scale Software Systems) and CNRS PICS project 07313 (SuCCeSS).

References

[1] R. M. Amadio. A synchronous pi-calculus. *Inf. Comput.*, 205(9):1470–1490, 2007.

[2] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

[3] J. Arias, M. Cano, and J. A. Pérez. A Reactive Interpretation of Session-Based Concurrency (Extended Version). URL: `http://www.mcanog.info/publications`.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[6] L. Bocchi, W. Yang, and N. Yoshida. Timed multiparty session types. In *Proc. of CONCUR'14*, volume 8704 of *LNCS*, pages 419–434. Springer, 2014.

[7] G. Boudol. ULM: A core programming model for global computing: (extended abstract). In *13th European Symposium on Programming, ESOP*, volume 2986 of *LNCS*, pages 234–248. Springer, 2004.

[8] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.

[9] M. Cano, C. Rueda, H. A. López, and J. A. Pérez. Declarative interpretations of session-based concurrency. In *Proc. of PPDP'15*, pages 67–78. ACM, 2015.

[10] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. In *Proc. of FSTTCS 2010*, volume 8 of *LIPIcs*, pages 338–351, 2010.

[11] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *Proc. of CONCUR 2008*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[12] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL 1987, Proceedings*, pages 178–188, 1987.

[13] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-adaptive multiparty sessions. *Service Oriented Computing and Applications*, 9(3-4):249–268, 2015.

[14] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.

[15] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proc. of PPDP'12*, pages 139–150. ACM, 2012.

[16] R. de Simone, J. Talpin, and D. Potop-Butucaru. The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook.* CRC Press, 2005.

[17] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[18] X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.

[19] A. Gamati. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer, 1st edition, 2009.

[20] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[21] N. Halbwachs. Synchronous programming of reactive systems. In *CAV'98*, volume 1427 of *LNCS*, pages 1–16. Springer, 1998.

[22] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.

[23] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-

Based Programming. In *Proc. of ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[24] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL'08*, pages 273–284. ACM, 2008.

[25] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, Apr. 2016.

[26] D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, 2016.

[27] H. A. López, C. Olarte, and J. A. Pérez. Towards a unified framework for declarative structured communications. In *PLACES 2009, York, UK, 22nd March 2009.*, volume 17 of *EPTCS*, pages 1–15, 2009.

[28] L. Mandel and C. Pasteur. Reactivity of cooperative systems - application to reactiveml. In *21st International Symposium, SAS 2014, Munich, Germany, 2014.*, volume 8723 of *LNCS*, pages 219–236. Springer, 2014.

[29] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proc. of PPDP'05*, pages 82–93. ACM, 2005.

[30] R. Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[31] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[32] M. Neubauer and P. Thiemann. An implementation of session types. In *PADL 2004, USA, June 18-19, 2004, Proceedings*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[33] D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In *POPL 2016, USA, 2016*, pages 568–581. ACM, 2016.

[34] L. Padovani. FuSe - A simple library implementation of binary sessions. URL: `http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html`.

[35] A. Scalas and N. Yoshida. Lightweight session programming in scala. In *Proc. of ECOOP 2016*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[36] Scribble: Describing multi party protocols. URL: `http://www.scribble.org/`.

[37] V. T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

[38] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The scribble protocol language. In *Proc. of TGC 2013*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.

[39] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.