Towards A Practical Model of Reactive Communication-Centric Software*

Jaime Arias[1], Mauricio Cano[2], and Jorge A. Pérez[2]

[1] University of Bordeaux, CNRS LaBRI UMR, INRIA, France
[2] University of Groningen, The Netherlands

Abstract. Many distributed software systems are *communication-centric*: they are composed of heterogeneous software artifacts that interact following precise communication structures (*protocols*). One much-studied approach to system analysis equips process calculi with *behavioral types* (such as *session types*) so to abstract protocols and verify interacting programs. Unfortunately, existing behaviorally typed frameworks do not adequately support *reactive behavior*, an increasingly relevant feature in protocols. To address this shortcoming, We have been exploring how the *synchronous programming* paradigm can uniformly support the formal analysis of reactive, communication-centric programs. In this short communication, we motivate our approach and report on ongoing developments.

1 Introduction

In this short note, we describe our ongoing work on a *reactive* programming model for *communication-centric* software systems. While most previous work relies on models based on the π-calculus [14], we are developing practical support for communication-centric software systems using ReactiveML [13], a *synchronous* programming language with functional and reactive features, and which relies on solid formal foundations.

In communication-centric software systems, collections of heterogeneous software artifacts usually follow well-defined communication structures, or *protocols*. Ensuring that programs conform to these protocols is key to certify system correctness. One much-studied approach to the analysis of communicating programs uses *behavioral types* [12], a type-based verification technique that captures complex communication structures while enforcing resource-usage policies. *Session*

* Research partially supported by EU COST Action IC1201 (Behavioral Types for Reliable Large-Scale Software Systems) and CNRS PICS project 07313 (SuCCeSS).

V. Biló, A. Caruso (Eds.): ICTCS 2016, Proceedings of the 17th Italian Conference on Theoretical Computer Science, 73100 Lecce, Italy, September 7–9 2016, pp. 227–233 published in CEUR Workshop Proceedins Vol-1720 at `http://ceur-ws.org/Vol-1720`

types [11] are a class of behavioral types that organize protocols as *sessions* between two or more participants; a session type describes the contribution of each partner to the protocol. First formulated as a type theory for the π-calculus, session-based concurrency has been implemented as communication libraries for mainstream languages, such as OCaml [15] and Scala [16].

One shortcoming of existing implementations is that they are based on overly rigid programming models. In particular, current practical support for communication-centric software systems does not explicitly consider *reactive behavior* in communicating programs. This is a crucial feature, especially as autonomous agents can now engage into protocols in our behalf (e.g., financial transactions). In fact, reactive behavior is central in realistic implementations of protocols with, e.g., exception handling, dynamic reconfiguration, and time. While these features can be represented in languages based on the π-calculus (cf. e.g., [6,3]), resulting models are often difficult or unnatural to reason about. Session types themselves focus on representing communication structures and thus abstract away from aspects related to reactivity. As protocols in emerging applications are increasingly subject to external stimuli/events (typically hard to predict), developing programming support that uniformly integrates structured communications and flexible forms of reactive behavior appears as a pressing need.

To our knowledge, the amalgamation of reactive behavior into models of structured communications has been little explored by previous works (see, e.g., [9]). Our efforts have been triggered by our declarative interpretation of session-based concurrency [5]. Our current work goes beyond the interpretation in [5] so to consider reactive and declarative behavior from a programming languages perspective. To this end, we have developed an implementation of sessions in ReactiveML [13], supported by a formal translation of session processes as ReactiveML programs. Based on our preliminary results, we believe that models of reactive programming improve previous works by offering a uniform basis for expressing and reasoning about different kinds of constructs.

2 Session Types

Session types offer a type-based methodology to the validation of communicating programs [11]. Structured dialogues (protocols) between interacting parties are represented as *sessions*; sequences of interactions along each channel in the program/process are then abstracted as types, which can be used to (statically) verify whether a program conforms to its intended protocols. Key properties are *fidelity* (programs respect prescribed protocols) and *communication safety* (programs do not have errors, e.g., communication mismatches). The syntax of (binary) session types T, S is as follows:

$!T.S$	Output a value of type T, continue as type/protocol S
$?T.S$	Receive a value of type T, continue as type/protocol S
$\&\{l_i : T_i\}_{i\in I}$	External choice among labeled types/protocols T_i (branching)
$\oplus\{l_i : T_i\}_{i\in I}$	Internal choice of a labeled type/protocol T_j, with $j \in I$ (selection)
$\mu X.T$	Recursive protocol/type (with type variable X)
end	Terminated protocol

In session-based concurrency, the notion of *duality* is key to ensure communication safety. Intuitively, duality relates session types with opposite behaviors: e.g., the dual of input is output, and vice versa; branching is the dual of selection, and vice versa.

We illustrate session types using a traditional example in the literature: the *Buyer-Seller-Shipper protocol*, which can be informally described 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 confirms to Shipper his availability for receiving the item.

We may formalize this protocol using the following session types:

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

where `item`, `confirmation`, `address`, and `ETA` denote basic types. Type *BuySell* describes interactions between Buyer and Seller from Buyer's perspective. Similarly, *SellShip* describes an interaction from Seller's perspective, and *ShipBuy* takes the standpoint of Shipper in communications. Complementary types can be obtained using duality. These three sessions take place in order, as in the informal description above.

3 A Reactive Approach to Communication-Centric Systems

The protocol presented before is well-suited for deployment using traditional technologies (e.g., web services). However, it does not consider the possibility of changes at runtime due to unexpected circumstances or external events. Moreover, the protocol is not suited to emerging scenarios in which protocol partners are deployed in, e.g., mobile devices with limited computational power and availability. For instance, it is easy to imagine Shipper being implemented by a drone with communication capabilities.

To address these shortcomings of protocol descriptions in session-based concurrency, we propose to use reactive behavior, as present in *synchronous reactive* *programming.* In this context, we can envision a reactive variant of the Buyer-Seller-Shipper protocol, in which Shipper is a drone, and Buyer communicates from a mobile phone. In this variant of the protocol, the first six steps are as before; after Steps 1–6, an *event* from Buyer to Seller triggers the following protocol:

R1. Buyer adds an item to his recently completed order.
R2. Seller replies back confirming the modified order.
R3. Seller forwards the modified order to Shipper.
R4. Shipper replies back in one of the following ways:
  a) Shipper returns back to the store, picks up the new item, and confirms to Buyer the previously given estimated delivery time, or
  b) Shipper continues with the original order, and informs Buyer that the second item will be delivered separately.

That is, in the reactive Buyer-Seller-Shipper protocol some of the exchanges are “standard” or “default” (cf. Steps 1-6); there are also other exchanges that are executed as a reaction to some event or external circumstance (cf. Steps R1-R4). In the latter steps, the external event concerns the request by Buyer of modifying his order; other conceivable conditions include, e.g., drone malfunctioning and wrong/delayed package deliveries. These extra exchanges also constitute structured protocols, amenable to specification and validation using sessions; however, their occurrence can be very hard to predict.

Synchronous Reactive Programming and ReactiveML. Synchronous Reactive Programming (SRP) is an event-based model of computation, optimized for programming reactive systems [1]. 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 computes new output values based on the input values and its internal state; the communication of all events between components occurs synchronously during each reaction. 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 makes it possible to reason about some time-related properties [8,10].

ReactiveML is an SRP-based extension to the OCaml programming language [13], based on the reactive model presented in [4]. This model 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 [2]. ReactiveML extends OCaml with the notion of *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 considered as instantaneous in ReactiveML.

In ReactiveML, synchronization is based on *signals*: events that occur in one logical instant. Signals can trigger reactions in processes, to be executed instantaneously or in the next time unit. Signals can carry values and can be emitted from different processes in the same logical instant. There are three basic ReactiveML constructs:

`emit s v`	emits signal `s` with value `v`.
`await one s <e> in P`	awaits a value along signal `s` that is pattern-matched to expression `<e>`. Process `P` is executed in the next instant.
`signal s in P`	declares signal `s` and bounds it to continuation `P`.

Our Current Work: Structured Communications in SRP. Models of communication-centric systems (such as session types) usually rely on directed exchanges along named channels. However, in SRP there is no native notion of channels: as we have seen, signals are the main synchronization unit in ReactiveML. To deal with this discrepancy, a key idea in our work is "simulating" channels using signals. To this end, and since we would like to represent protocols respecting linearity, we follow the representation of session channels in [7], which uses a continuation-passing style. This means that for each interaction within a communication structure a new channel is created.

We describe our ReactiveML implementation for Seller in the reactive Buyer-Seller-Shipper protocol. Recall that Seller is involved in two sessions: he first communicates with Client, then interactions with Shipper occur. In the code below we assume that all sessions have been already initiated; these are noted `cb` for Buyer and `cs` for Seller.

```
let process seller conf =
  await one cb (item,y) in signal c1 in
  emit y (conf,c1);pause;
  await one c1 (addr,u) in signal c2 in
  emit cs (addr,c2);pause
```

The code declares `seller` as a process in ReactiveML (a non-instantaneous function); we describe its body. The first line awaits a signal `cb`, which carries a pair of elements: a value and a reference to the signal where further interactions will occur (i.e., `y`). Then, a signal `c1`, where the next interaction will occur, is declared. Subsequently, a pair (containing a message `conf` and a reference to signal `c1`) is emitted over signal `y`; no further actions are taken in this time unit. Once the message is received by Buyer, `seller` awaits Buyer's address. At this point, the first session has finished and communication with Shipper begins. In the last line, Buyer's address is sent to Shipper.

Notice that once `seller` is finished, so is any communication from Seller. But in the reactive protocol, Seller must await possible further actions from either Buyer or Shipper. To implement this key feature, we extend the previous code with the following line of code: `await one g (v,e) in P`. This new line puts `seller` to "sleep" until an event/signal `g` from either Buyer or Seller triggers a new reaction `P` from it. Note that this signal for *interrupts* or events should be known to every party in the communication. The idea is then that the continuation process `P` should decide which course of action to take depending on the value carried by `g`. In our reactive protocol, process `P` could implement Steps R1-R4, following the implementation scheme of `seller`.

4 Concluding Remarks

We have described our ongoing implementation of essential features of session-based concurrency in ReactiveML, a reactive functional programming language. Our implementation uses ReactiveML processes to handle usual session protocols (send, receive, select, and branch constructs). We believe that our approach is already on a par with other session implementations (such as [15]) with substantial room for improvement, due to the reactive behavior supported by ReactiveML. We expect our research to enable the practical use of session-based concurrency into emerging application scenarios, such as, e.g., Collective Adaptive Systems (CAS).

References

1. 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.
2. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
3. L. Bocchi, W. Yang, and N. Yoshida. Timed multiparty session types. In *Proc. of CONCUR'14*, volume 8704, pages 419–434. Springer, 2014.
4. F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.
5. 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.
6. M. Carbone. Session-based choreography with exceptions. *Electr. Notes Theor. Comput. Sci.*, 241:35–55, 2009.
7. O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proc. of PPDP'12*, pages 139–150, 2012.
8. R. de Simone, J. Talpin, and D. Potop-Butucaru. The synchronous hypothesis and synchronous languages. In *Embedded Systems Handbook*. CRC Press, 2005.
9. 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.
10. A. Gamati. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification.* Springer, 1st edition, 2009.
11. 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, pages 122–138. Springer, 1998.
12. 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.

13. L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proc. of PPDP'05*, pages 82–93. ACM, 2005.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
15. L. Padovani. FuSe - A simple library implementation of binary sessions. URL: `http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html`.
16. A. Scalas and N. Yoshida. Lightweight session programming in scala. In *ECOOP 2016*, LIPIcs. Dagstuhl, 2016.