STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXI**, Number 1, 2016

COMPARISON OF SESSION LOGIC WITH SESSION TYPES

TIBOR KISS

Abstract. The aim of this paper is to compare two states of the art techniques of protocol verification, namely: *Session Types* and *Session Logic*, in terms of their applicability in an industrial environment. The evaluation was done by modelling a set of industrial protocols with both methods, and comparing them from a qualitative point of view. For comparison, we considered the following qualitative properties of the encoded protocols: the specification expressiveness in the encoding of safety and functional requirements, the efficiency of the protocol from data transmission point of view and the re-usability of the specification. The results of this comparison are summarised in three business protocol examples which are presented in detail in this paper. Despite the fact that the two formalism present minor differences theoretically, the experimental results showed that the difference between the two techniques is significant.

1. Introduction

Modern programming concepts [5, 7, 17, 23] are based on the assumption that one of the primary and fundamental aspects of modern programs is communication. This assumption is underlined by the fact that a lot of challenging aspects of programming (like the processes synchronization, access to persistent storage systems, user interaction, the access of shared resources and data exchange between processes) can be modelled (and implemented) with communication.

A simple technique for addressing this issue is process calculus which provides a family of approaches for formal modelling of structured interaction, providing a set of tools to describe the high-level communications between a collection of independent processes. Examples of process calculi include CSP (Communicating Sequential Processes) [19], CCS (Calculus of Communicating

Received by the editors: January 12, 2016.

1998 *CR Categories and Descriptors.* D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification – *Formal methods*; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors – *Code generation*.

Key words and phrases. Proof-based development, Program Verification, Session types, Session Logic, Separation Logic.

Systems) [22] and π - calculus [21]. These calculi provide a set of algebraic laws that allow formal reasoning about equivalences between processes.

Additionally, due to its importance, a number of researchers have focused on the problems of constraining the shape of processes by way of type systems. However, the direct application of the theoretical typing techniques to the mainstream engineering languages, presents a few obstacles. Existing type systems are targeted at calculi with first-class primitives for linear communication channels and communication oriented control flow. The majority of mainstream engineering languages needs to be extended in this sense to be suitable for syntactic session type checking. As an answer to this problem, a set of revolutionary new theories appeared in the last few years to enforce the correctness of the processes implemented in a mainstream programming languages via logic.

The object of this paper is to study the precise relationship between these two formalism. In particular, for type system, we choose session types as a widely studied formalism in this area, and we compare with session logic, a novel formalism to verify protocol specification correctness.

The comparisons of the superficially different formalisms enlightening common underlying concepts, will hopefully improve the language design and the programming practice for communication based computing.

In this paper, we will survey all these aspects informally, by means of examples. These examples are based on our experiments with the HIP/SLEEK extension for session logic[1]. We begin in Section 2 with session types in their global versus local formulation, where the basic concepts and formalisms are presented. Section 3 presents the session logic as an improvement of session types which have been proposed to gain expressiveness and to catch stronger computational properties. Section 4 is devoted to compare with examples the two formalisms into an imperative and object-oriented programming paradigms and finally in Section 5, we quickly review the two formalisms in the conclusion.

2. Session types

Session types are a type formalism used to model structured communication-based programming for distributed systems [12]. In particular, binary session types, describe the communication between exactly two participants in such scenarios [8]. When session types are added to a standard communication channel, it must statically enforce that client-server communication proceeds according to a previously defined choreography. The syntax of session types is illustrated in Fig. 1, where S is the set of closed session type terms, then the

[1], which is a recently developed extension by us to verify session logic

syntax can be interpreted as follows: type *begin* is the type of an unopened session channel; end is a terminated session channel; $K!\langle T\rangle;S$ and $K?\langle T\rangle;S$ indicate respectively the types for sending and receiving a message of type T and continuation of session type S.

Select and branch, $K \oplus \{l_i : S_i\}_{i\in I}$ and $K\&\{l_i : S_i\}_{i\in I}$ are sets of labelled session types indicating, respectively, external and internal choice. $\mu s.S$ and s model recursive session types.

Usually, this typing discipline includes also a duality function which constructs a specific dual type for any given session type. The definition of inductive duality can be found in Fig. 2.

S ::=	$K!\langle T\rangle;S$	send
	$K?\langle T\rangle;S$	receive
	$K \oplus \{l_i : S_i\}_{i\in I}$	selection
	$K\&\{l_i : S_i\}_{i\in I}$	branching
	$\mu s.S \mid s$	recursion
	begin — end	begin, end

FIGURE 1. Binary Session Types.

The duality is an important part of the theory of session types because allows us to verify both ends of a communication channel, using the same specification.

$$\overline{K!\langle T\rangle.S} = K?\langle T\rangle.\overline{S} \qquad \overline{K?\langle T\rangle.S} = K!\langle T\rangle.\overline{S}$$
$$\overline{K \oplus \{l_i : S_i\}_{i\in I}.S} = K\&\{l_i : \overline{S_i}\}_{i\in I}.\overline{S} \qquad \overline{K\&\{l_i : S_i\}_{i\in I}.S} = K \oplus \{l_i : \overline{S_i}\}_{i\in I}.\overline{S}$$
$$\overline{\mu s.S} = \mu s.\overline{S} \quad \overline{s.S} = s.\overline{S} \qquad \overline{begin.S} = begin.\overline{S} \quad \overline{end.S} = end.\overline{S}$$

FIGURE 2. Session Types Dual Specification.

The semantics of session types is defined in terms of a subtyping relation. A detailed definition of this semantic can be found in [1].

3. Session Logic

As far as we know, the session logic from [11] is the first to introduce a dedicate logical theory which enables effective compile-time assertion-based validations of protocol specification for a typed imperative program.

Different from previous approaches, session logic proposes a novel use of disjunction to specify and verify the implementation of communication protocols. Even though the logic is based on two-party channel sessions, it can also handle delegation through the use of higher-order channels. Furthermore, due to the use of disjunctions to model both internal and external choices, we need to use only conventional conditional statements to support both kinds of choices. In contrast, session types require the host languages to be extended with a set of specialized switch constructs to model both internal and external choices.

$$
\begin{array}{lll}
\mathit{spred} & ::= p(\mathtt{root}, v^*) \equiv \Phi \quad\quad \Phi ::= \bigvee \sigma^* \quad\quad \sigma ::= \exists\ v^* {\cdot} \kappa {\wedge} \pi \\
\mathit{mspec} & ::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po}; \\
S & ::= \mathtt{emp} \mid ?r \cdot \Phi \mid !r \cdot \Phi \mid \sim S \mid S_1; S_2 \mid S_1 \vee S_2 \\
\Delta & ::= \Phi \mid \Delta_1 {\vee} \Delta_2 \mid \Delta {\wedge} \pi \mid \Delta_1 {*} \Delta_2 \mid \exists v {\cdot} \Delta \\
\kappa & ::= \mathtt{emp} \mid v {\mapsto} c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 \mid \mathcal{C}(v, S) \quad \pi ::= \gamma \wedge \phi \\
\gamma & ::= v_1{=}v_2 \mid v{=}\mathtt{null} \mid v_1{\neq}v_2 \mid v{\neq}\mathtt{null} \mid \gamma_1{\wedge}\gamma_2 \\
\phi & ::= r : t \mid \varphi \mid b \mid a \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= \mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2 \quad\quad a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}}{\times}s \mid s_1{+}s_2 \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \mid |\mathtt{B}| \\
\varphi & ::= v{\in}\mathtt{B} \mid \mathtt{B}_1{=}\mathtt{B}_2 \mid \mathtt{B}_1{\sqsubset}\mathtt{B}_2 \mid \mathtt{B}_1{\sqsubseteq}\mathtt{B}_2 \mid \forall v{\in}\mathtt{B}{\cdot}\phi \mid \exists v{\in}\mathtt{B}{\cdot}\phi \\
\mathtt{B} & ::= \mathtt{B}_1{\sqcup}\mathtt{B}_2 \mid \mathtt{B}_1{\sqcap}\mathtt{B}_2 \mid \mathtt{B}_1{-}\mathtt{B}_2 \mid \emptyset \mid \{v\}
\end{array}
$$

FIGURE 3. Session Logic Specification Language.

Additionally, session logic is based on an extension of separation logic, and thus it supports heap-manipulating programs and copyless message passing.

As channels can support a variety of messages, the read content can be treated as dynamically typed where conditionals are dispatched based on the received types. Alternatively, type-safe casting can be guaranteed via the verification of communication safety. In addition, Session Logic can go beyond such cast safety by ensuring that heap memory and properties of values passed into the channels are suitably captured.

The specification language in Fig. 3 allows shape predicates *spred* to specify program properties in a combined domain. Note that such predicates are constructed with disjunctive constraints Φ.

$$
\begin{array}{ll}
\sim !r \cdot \Delta = ?r \cdot \Delta & \sim ?r \cdot \Delta = !r \cdot \Delta \\
\sim(S_1 \vee S_2) = \sim S_1 \vee \sim S_2 & \sim(S_1; S_2) = \sim S_1; \sim S_2
\end{array}
$$

FIGURE 4. Rules for Dual Specification.

A session specification for channel v is represented by $\mathcal{C}(\mathtt{v}, \mathtt{S})$ where S can denote a sending communication, a receiving communication, a sequence of communication operations and a choice of communication operations. S can also capture pure (e.g. type) or heap properties of the exchanged messages. An abstract program state σ has mainly two parts: the heap part and the pure part and it is extended with several domains as: bag domain, integer domain and the new session logic domain. During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ. An abstract state Δ can be normalized to the Φ form [10]. The rules to obtain dual specifications are given in Fig. 4.

4. Examples

We start our comparison by extending Fig. 1 from [14], which is a business protocol example between Buyer, Shipper and Seller.

From the beginning, the Buyer sends the product name as a `String` object to the Seller. The Seller replies by sending the product's price as a `double`. If Buyer is satisfied with the price, she sends a strict positive quantity as an integer to the Seller, otherwise it sends zero and quits the conversation. If the Buyer buys the product then the Seller establishes a connection with the Shipper in order to arrange the transportation of the product. The Seller provides the necessary information about the product and also delegates the Buyer connection to the Shipper. Finally, the Shipper and the Buyer establishes the necessary detail related to the transportation. As part of this process, the Buyer provides to the Seller her address, and the Seller provides a delivery date to the Buyer. The example from Fig. 5 is a 3-party session but can be modeled as two 2-party sessions. In a 2-party session, one channel specification is typically sufficient for describing the communication between two parties. We will provide an incomplete model of the Buyer-Seller and Seller-Shipper protocols by using the following session types to represent the Buyer's and Shipper's communication pattern:

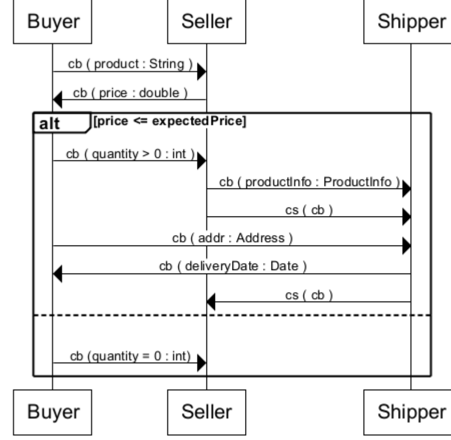

FIGURE 5. Buyer-Seller protocol

```
buyer_ty    ≡ begin.!String.?double.! <!int.!Addr.?Date.end, !int.end >
deleg_ty    ≡ !Addr;.?Date.end
shipper_ty  ≡ begin.?ProductInfo.?S(deleg_ty).!S(end).end
```

The first types specification is not accurate enough conform to the problem requirement, because the quantity and the choice data must be interconnected (the quantity must be greater than zero if the condition in the `sendIf` statement is true), so erroneous implementations such as from Fig.7 can not be captured. More detailed specification of the protocol is not possible in the existing session types formalism.

The dual specifications of the above session types correspond to the Seller's communication pattern, which are:

```
seller_buy_ty   ≡ begin.?String.!double.? <?int.?Addr.!Date.end, ?int.end >
seller_ship_ty  ≡ begin.!ProductInfo.!S(deleg_ty).?S(end).end
```

The program from Fig.6 that implements the above protocol uses specialized branching constructs, like `sendIf` and `receiveIf` to model the internal and external choices. Additionally, the program transmits unnecessarily a

boolean value representing the decision of the `sendIf`. This value which enlarges the transmitted data size is useless because the condition can be encoded into quantity, according to the specification.

```
void buyer(buyer_ty c, String p)
{ send(c, p);
  double price = receive(c);
  double budget = ...;
  sendIf (price <= budget)then{
    int q = ...;
    send(c, q);
    Addr a = ...;
    send(c, a);
    ShipDate sd = receive(c);
    send(c, 3);
}
```

```
void seller(seller_buy_ty cb,
        seller_ship_ty cs)
{ String p = receive(cb);
  send(cb, getPrice(p));
  receiveIf(cb) {
    int q = receive(cb);
    ProductInfo pi = ...;
    send(cs, pi);
    sendS(cs, cb);
    cb = receiveS(cs);
  }
}
```

```
void shipper(shipper_ty c)
{ ProductInfo pi;
  pi = receive(c);
  deleg_ty cs;
  cs = receiveS(c);
  Addr a = receive(cs);
  ShipDate sd = ...;
  send(cs, sd);
  sendS(c, ss);
}
```

FIGURE 6. Session Types Business Protocol Implementation

```
void buyer(buyer_ty c, String p)
{ send(c, p);
  double price = receive(c);
  int quantity = ...;
  sendIf (quantity == 0)then{
    send(c, quantity);
    Addr a = ...;
    send(c, a);
    ShipDate sd = receive(c);
    send(c, 3);
}
```

```
void buyer(buyer_ty c, String p)
{ send(c, p);
  double price = receive(c);
  double budget = ...;
  sendIf (price <= budget)then{
    int q = 0;
    send(c, q);
    Addr a = ...;
    send(c, a);
    ShipDate sd = receive(c);
    send(c, 3);
}
```

FIGURE 7. Business Protocol Erroneous Implementation

For the session logic-based approach, the above communication patterns for Buyer, Shipper and Seller could be represented, as follows:

$$
\begin{array}{lcl}
\texttt{buyer_ch} & \equiv & !\texttt{String}; ?\texttt{double}; ((!\texttt{r}{:}\texttt{int} \cdot \texttt{r}{>}0; !\texttt{Addr}; ?\texttt{Date}) \vee !0) \\
\texttt{seller_buy_ch} & \equiv & \sim\texttt{buyer_ch} \\
 & \equiv & ?\texttt{String}; !\texttt{double}; ((?\texttt{r}{:}\texttt{int} \cdot \texttt{r}{>}0; ?\texttt{Addr}; !\texttt{Date}; ?\texttt{int}) \vee ?0) \\
\texttt{deleg_ch} & \equiv & !\texttt{Addr}; ?\texttt{Date}; \texttt{end} \\
\texttt{shipper_ch} & \equiv & ?\texttt{ProductInfo}; ?\texttt{r}{:}\texttt{Chan} \cdot \texttt{C}(\texttt{r}, \texttt{deleg_ch}); !\texttt{r}{:}\texttt{Chan} \cdot \texttt{C}(\texttt{r}, \texttt{emp}) \\
\texttt{seller_ship_ch} & \equiv & !\texttt{ProductInfo}; !\texttt{r}{:}\texttt{Chan} \cdot \texttt{C}(\texttt{r}, \texttt{deleg_ch}); ?\texttt{r}{:}\texttt{Chan} \cdot \texttt{C}(\texttt{r}, \texttt{emp})
\end{array}
$$

Superficially, these logical specifications look similar to the previous session types; however, there are several notable differences. Firstly, there is no need

for any begin/end declarations since the protocol is expected to be locally captured after creation. Secondly, the logic makes use of disjunction[2] instead of some specialized notations for internal and external choices. This allows us to directly use conditionals to support choices which are naturally modelled by disjunctive formulae during program reasoning.

Thirdly, instead of transmitting the decision of the internal choice (as true or false in this case), we may just use values (such as greater than 0 or 0).

There are two benefits of this encoding. First, in contrast with session types this theory allows the verification of optimal protocols. [3] Second, the theory allows the expressing of functional properties into the protocol. Such a well-defined functional property from the previous specification is the encoding of the internal and external choices into quantity. The perfect encoding of this functional requirement allows a more precise verification of the implementation excluding errors like in Fig.7.

Most importantly, instead of types or values, the logic allows more general properties to be passed into the channel to facilitate the verification of functional correctness properties, which can go beyond communication safety. This includes the use of higher-order channels to model session types delegation, where channels and their expected specifications are passed as messages.

As a simple illustration, we may strengthen channel specification by using positive integers instead of merely integer prices. This change is captured by the following modified channel specification for Buyer.

$$
\begin{array}{lcl}
\texttt{buyer_chan} & \equiv & !\texttt{String}; !\texttt{r:double} \cdot \texttt{r}>0; ((!\texttt{r:int} \cdot \texttt{r}>0; !\texttt{Addr}; ?\texttt{Date}) \vee !0) \\
\texttt{seller_buy_chan} & \equiv & \sim\texttt{buyer_chan} \\
 & \equiv & ?\texttt{String}; !\texttt{r:double} \cdot \texttt{r}>0; ((?\texttt{r:int} \cdot \texttt{r}>0; ?\texttt{Addr}; !\texttt{Date}) \vee ?0)
\end{array}
$$

The specification `seller_buy_chan` is the dual specification of `buyer_chan`. Such dual specification is obtained by inverting the polarity of messages, where input is converted to output and vice-versa as specified in Fig.4.

Session logic also supports separation formulae for pointer-based message passing for shared memory implementation. Another issue worth noting is that thread specification and channel specification needs to be different. As an example, let us specify a stronger specification for seller's communication with the protocol, by insisting that price of products sold by this seller is at least 20 units, as follows:

[2]To support unambiguous channel communication, the disjunction by the receiver must have some disjoint conditions, so that the session logic may guarantee its synchronization with the `sender`.

[3]The `buyer_ty` protocol specification is not optimal because requires the transmission of the internal choice decision, in contrast with the specification`buyer_ch` which is optimal.

$$\texttt{seller_sp} \equiv \texttt{?String}; \texttt{!r:double} \cdot \texttt{r}{>}20; ((\texttt{?r:int} \cdot \texttt{r}{>}0; \texttt{?Addr}; \texttt{!Date}; \texttt{!int}) \vee \texttt{?0})$$

With this change, we can write a program that implements the above protocol, as shown in Fig. 8. Note that we can directly use conditionals instead of the specialized switch constructs as in Fig. 6.

The benefit of this change is twofold. Firstly, the verification mechanism can be applied to mainstream engineering languages, without the need to extend it with communication primitives like in the case of session types. Secondly, we can have an optimal implementation from the transmitted data perspective because the internal and external choices can be based on the transmitted data as in the `seller` function from Fig. 8.

```
open(cb) with buyer_chan;
open(cs) with shipper_chan;
(buyer(cb, prod) || seller(cb, cs) || shipper(cs));
close(cb);
close(cs);
```

```
void buyer(Chan c, String p)
  requires C(c, buyer_ch)
  ensures C(c, emp)
{ send(c, p);
  double price = receive(c);
  double budget = ...;
  if price <= budget then{
    int q = ...;
    send(c, q);
    Addr a = ...;
    send(c, a);
    ShipDate sd = receive(c);
    send(c, 3);
  } else send(c, 0); }
```

```
void seller(Chan cb, Chan cs)
 requires C(cb, seller_buy_ch)
       *C(cs, seller_ship_ch)
 ensures C(c, emp) * C(c, emp)
{ String p = receive(cb);
  send(cb, getPrice(p));
  int q = receive(cb);
  if q > 0 then {
    int q = receive(cb);
    ProductInfo pi = ...;
    send(cs, pi);
    send(cs, cb);
    cb = receive(cs);
  } }
```

```
void shipper(Chan c)
  requires C(c, shipper_ch)
  ensures C(c, emp)
{ ProductInfo pi;
  pi = receive(c);
  deleg_ty cs;
  cs = receive(c);
  Addr a = receive(cs);
  ShipDate sd = ...;
  send(cs, sd);
  send(c, cs);
}
```

FIGURE 8. Session Logic Business Protocol Implementation

Another important aspect is that the `seller` process specification `seller_sp` imposes a stronger property over the sent price, using `r>20` instead of `r>0` that is similar to the session types subtyping, but is more flexible.

In the end of this example, we want to emphasize the delegation as one of the most important distinctions between session types and other communication calculus-based methods. Without the enforcement of delegation, we cannot compare a verification method with session types. From delegation perspective the session logic specification is precise as session types and highlights the state of the transmitted channel, which must be insured by the

sender and can be assumed by the receiver, but unlike session types solution, session logic uses the same send/receive channel methods for sending values, data structures, and channels as can be seen in Fig.8.

In order to have a more precise comparison, we must take into consideration the example from [15]. The example is based on a business protocol between a buyer and a seller. The buyer has a choice between request a quote, accept a quote and quit. In the first case, she must send the name of a product, and then receive the price and a reference number for the quote. In the second case when the buyer wants to buy a product, she must send a quote reference followed by payment information. The problem is underspecified because it is not possible to buy an item before obtaining a quote. The session types and session logic specifications of this problem are the following:

```
buyer_r_ty   ≡  &{reqQuote :!String.?double.?OfferId.buyer_r_ty,
                  accQuote :!OfferId.!Payment.buyer_r_ty, quit.end}
buyer_ty     ≡  begin.buyer_r_ty
c_buyer_ty   ≡  begin.reqOffer.accOffer.end

req_ch       ≡  !String; ?double; ?Quote
buyer_r_ch   ≡  {!1; req_ch; buyer_r_ch∨!2; !Quote; !Pay; buyer_r_ch∨!3}
buyer_ch     ≡  req_ch; buyer_r_ch
```

The session types theory propose the collection of the `buyer_ty` specification into the `c_buyer_ty` class session type instead of annotating the method definitions with pre- and post-conditions as in session logic. This approach has several advantages and disadvantages. The benefit is that we have an abstract behaviour model for each class, which allows the modular verification of the usage of classes and also the effective encapsulation of channels in objects. On the other hand, this typing discipline is at a disadvantage compared to session logic when it comes to functional correctness and flexibility. This logic allows also the modular verification of communication properties (as can be seen in Fig.9), but in contrast with session types, the methods are reusable (see Fig.1 from [15]) and the functional correctness can be enforced.

```
void req(Chan c, String p)
requires C(c, req_ch; rest)
ensures C(c, rest)

void pay(Chan c, Payment p)
requires C(c, !2; !Quote; !Pay; rest)
ensures C(c, rest)
```

```
// C(c, buyer_ch)
req(c, prod_name);
// C(c, buyer_r_ch)
pay(c, pay_inf);
// C(c, buyer_r_ch)
send(c, 3);
// C(c, emp)
```

```
// C(c, buyer_ch)
pay(c, pay_inf);
// VerificationError!!
send(c, 3);
// Invalid code!!
```

FIGURE 9. Session Logic Business Protocol Implementation

In the end, we highlight the expressiveness of session logic, by using a new business protocol example between Buyer and Seller. The Buyer recursively sends a read-only list of product identifiers, while the Seller responds with a

price for each product identifier. The sequence diagram of the problem can be found in Fig.10.

Given the following data node declaration and linked list definition:

$$\texttt{data node}\{\texttt{int id};\ \texttt{node next};\} \quad \texttt{pred ll(root)} \equiv \texttt{root} = \texttt{null} \vee \exists\, \texttt{q} \cdot \texttt{root}{\mapsto}\texttt{node}\langle _, \texttt{q}\rangle * \texttt{ll(q)}$$

The communication specification between buyer and seller can be written using an inductive definition, as below:

$$\texttt{buy_lsp} \equiv\ !\texttt{p}:\texttt{node} \cdot \texttt{p} = \texttt{null} \vee\ !\texttt{p}:\texttt{node} \cdot \texttt{p}{\mapsto}\texttt{node}\langle _, _\rangle; ?\texttt{Double}; \texttt{buy_lsp}$$

The protocol specification asserts that each outward transmission of a not `null` `node` must be followed by an input of type `Double`. The communication terminates once the Buyer has received a null reference from the seller, which marks the end of the list.

The fact that the communication uses node transmission serves a double scope: for sharing product information and for ensuring that the Buyer's loop and the Seller's loop are synchronized. As opposed to other session types enforcement techniques, the synchronization of the loops are done via the transmitted data. Generally, the session type techniques require a flag transmission at each iteration in order to ensure that the loops have same iterations. In contrast, session logic allows the verification of a more optimal implementation by using separation logic to specify and verify this example.

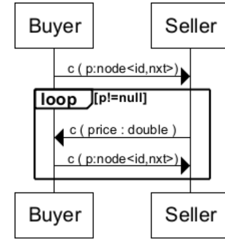

Figure 10. List example

```
void buyer(Chan c)
  requires C(c, buy_lsp)
  ensures C(c, emp)
{ node it = getItems();
 recvPrices(c, it);}

void seller(Chan c)
  requires C(c, ~buy_lsp)
  ensures C(c, emp)
{ node it = receive(c);
 if(it! = null){
 send(c, price(it.id));
 freeNode(it);
 seller(c);}}
```

```
void recvPrices(Chan c, node it)
  requires C(c, buy_lsp) * ll(it)
  ensures C(c, emp)
{ if(it! = null){
 node nxt = it.next;
 int id = it.id;
 send(c, it);
 Double price = receive(c);
 procPrice(id, price);
 recvPrices(c, nxt);
 } else {
 send(c, it);}
}
```

Figure 11. Items Purchasing implementation

5. Conclusion

The article compares two states of the art theories which try to enforce the protocol specification via verification. Due to space limitations, we focus on the key differences between the two formalisms.

First, session types proposals require the host language to be extended with a set of specialized linear communication primitives like `send`, `receive` and `internal`- and `external` − `choice` [15, 18, 13], therefore their type theory cannot be applied to industrial mainstream languages. This issue is well known in the literature, therefore, are several works which enforce the session types specification via dynamic verification [9, 2, 20, 24, 16]. To handle the problem, session logic uses disjunctions to model both internal and external choices, in consequence, the hosting language can use conventional conditional statements to support both kinds of choices.

Second, session types are not flexible enough to encode size optimized protocols, [4] due to its limitation to have external choices based on a transmitted value. In contrast, session logic which uses logical formulas to model the external choices allows such flexible encoding.

Third, the weak constraint constituted by typing channels with session types is not always sufficient to detect subtle communication errors [3]. These are caused by the fact that, viewed as constraints on behaviour, session types have a much less restrictive power than session logic.

In addition, two downsides of session logic must be mentioned. First, there are situations where the session logic entailment is undecidable. Second, theoretically the logic is not so founded as session types (some aspect of verification like exception handling [6], time constraints [4] and multiparty session types [17] are not developed).

Despite their previous drawbacks, this, apparently small change, constituted by verifying channels with logic, is sufficient to detect subtle errors in industrial mainstream languages. In fact, it reveals to be the right setting where concepts of deductive verification (like Hoare logic or separation logic) for imperative program verification can be combined, for functional verification of distributed systems.

6. Acknowledgements

This work is supported by Siemens grant no. 7472/3202246933.

[4] An example that shows this problem is presented in section 4.

References

[1] Giovanni Bernardi, Ornela Dardha, Simon J Gay, and Dimitrios Kouzapas. On duality relations for session types. In *Trustworthy Global Computing*, pages 51–66. Springer, 2014.

[2] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.

[3] Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. A multiparty multi-session logic. In *Trustworthy Global Computing*, pages 97–111. Springer, 2013.

[4] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR 2014*. Springer, 2014.

[5] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *Coordination Models and Languages*, pages 49–64. Springer, 2014.

[6] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR 2008-Concurrency Theory*, pages 402–417. Springer, 2008.

[7] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 263–274. ACM, 2013.

[8] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of session types. ACM, 2009.

[9] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *Trustworthy Global Computing*, pages 25–45. Springer, 2012.

[10] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated Verification of Shape, Size and Bag Properties Via User-Defined Predicates in Separation Logic. *Sci. of Comp. Prog.*, 77:1006–1036, 2012.

[11] Florin Craciun, Tibor Kiss, and Andreea Costea. Towards a session logic for communincation protocols. In *International Conference on Engineering of Complex Computer Systems, Australia*, 2015.

[12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 139–150, New York, NY, USA, 2012. ACM.

[13] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*. Springer Berlin Heidelberg, 2007.

[14] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. Springer Berlin Heidelberg, 2006.

[15] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. *SIGPLAN Not.*, 45(1):299–312, January 2010.

[16] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Denilou, and Nobuko Yoshida. Structuring communication with session types. In *Concurrent Objects and Beyond*. Springer Berlin Heidelberg, 2014.

[17] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 2008.

[18] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *ECOOP 2010*. Springer Berlin Heidelberg, 2010.

[19] Harold D Lasswell. The structure and function of communication in society. *The communication of ideas*, 37:215–228, 1948.

[20] Eduardo R. B. Marques, Francisco Martins, Vasco Thudichum Vasconcelos, Nicholas Ng, and Nuno Dias Martins. Towards deductive verification of mpi programs against session types. Open Publishing Association, 2013.

[21] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[23] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[24] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local verification of global protocols. Springer Berlin Heidelberg, 2013.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca

E-mail address: `kisst@cs.ubbcluj.ro`