Towards a Session Logic for Communication Protocols

Florin Craciun, Tibor Kiss
Faculty of Mathematics and Computer Science,
Babes-Bolyai University, Cluj-Napoca
Email: {craciunf, kisst}@cs.ubbcluj.ro

Andreea Costea
School of Computing,
National University of Singapore
Email: andreeac@comp.nus.edu.sg

***Abstract*—Communication-centered programming is one of the most challenging programming paradigms. Development of modern software applications requires expressive mechanisms to specify and verify the communications between different parties. In the last decade, many works have used session types to characterize the various aspects of structured communications. Different from session types, we propose a novel session logic with disjunctions to specify and verify the implementation of communication protocols. Our current logic is based on only two-party channel sessions, but it is capable of handling delegation naturally through the use of higher-order channels. Due to our use of disjunctions to model both internal and external choices, we rely solely on conditional statements to support such choices, as opposed to specialized switch constructs in prior proposals. Furthermore, since our proposal is based on an extension of separation logic, it also supports heap-manipulating programs and copyless message passing. We demonstrate the expressivity and applicability of our logic on a number of examples.**

I. Introduction

With the success of tools like [1], [2] and [3] for the development and verification of non-concurrent systems, there are now deeper desires for similar tools for distributed systems. Additionally, the communication requirement, which is ubiquitous in software systems from the information exchange between software entities to the communication of these systems with their environments, must be properly verified to affirm the system's correctness. Due to its importance, a number of researchers have focused on the problems of ensuring safe communication in the last few decades.

CSP (Communicating Sequential Processes) [4] and CCS (Calculus of Communicating Systems) [5] are among the earliest theories to address the communication problems. The most recent extensions for these works are based on session types, and their derivatives, such as contracts [6]. In the last decade, session types have been integrated into a number of programming languages and process calculi, including functional languages [7], [8], object-oriented languages [9], [10], calculi of mobile processes [11], and higher-order processes [12]. Recently, session types have also been extended with logic [13] to act as a contract between the communication entities. This extension allows a more precise verification of the involved parties by enabling a concise specification of the transmitted messages on what one party must ensure, and from which the other party can rely on it. There was also a proposal for multi-party session logic [14], but this logic tries to also summarize the effects of processes involved in the protocol. In contrast, we propose a session logic which focuses entirely on the communication patterns, while the effects of the associated processes are summarized directly in each thread's pre- and postcondition.

Contributions: Different from previous approaches, we propose a session logic with a novel (and natural) use of disjunction to specify and verify the implementation of communication protocols. Even though the currently proposed logic is based on two-party channel sessions, it can also handle delegation through the use of higher-order channels. Unlike past solution on delegation [10], our proposal uses the same send/receive channel methods for sending values, data structures, and channels. For example, [10] requires a separate set of send/receive methods to support higher-order channels. Furthermore, due to our use of disjunctions to model both internal and external choices, we need only use conventional conditional statements to support both kinds of choices. In contrast, past proposals typically require the host languages to be extended with a set of specialized switch constructs to model both internal and external choices. Additionally, our proposal is based on an extension of separation logic, and thus it supports heap-manipulating programs and copyless message passing. Lately, Villard et al. [15] have designed a logic for copyless message passing communication. Their logic relies on state-based global contracts while our more general logic of session is built as an extension of separation logic with disjunction to support communication choices. The logical formulae on protocols can also be localised to each channel and may be freely passed through procedural boundaries. Villard et al. [15] currently use double-ended channels to solely support communication safety, but do not guarantee deadlock freedom. In contrast, a channel in our proposal is multi-ended with its complementary properties captured in local specifications, which are supplied via each of the channel's aliases. As channels can support a variety of messages, we can treat the read content as dynamically typed where conditionals are dispatched based on the received types. Alternatively, we may also guarantee type-safe casting via verifying communication safety. We can also go beyond such cast safety by ensuring that heap memory and properties of values passed into the channels are suitably captured. Lastly by using a subsumption relation on our communication proposal, we allow specifications on channels to differ between threads but would ensure that they remain compatible at each join point, in order to prevent intra-channel deadlocks. More realistically, we also assume the presence of asynchronous communication protocols, where send commands are non-blocking.

In this paper, we argue strongly on the simplicity, expressivity and applicability of our logic by demonstrating it through a number of examples.

II. A Motivating Example

We introduce our session logic-based approach by using a simple business protocol example between Buyer 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 an `int`. If Buyer is satisfied with the price, she sends the address as an object of type `Addr` and Seller sends back the delivery date as an object of type `Date` (the definitions of these types are presented in the appendix A). Otherwise, the Buyer quits the conversation. This example is modeled as 2-party session in Fig. 1. In a 2-party session, one channel is typically sufficient for communication between two parties. We can summarize this Buyer-Seller protocol by using the following session type to represent the Buyer's communication pattern:

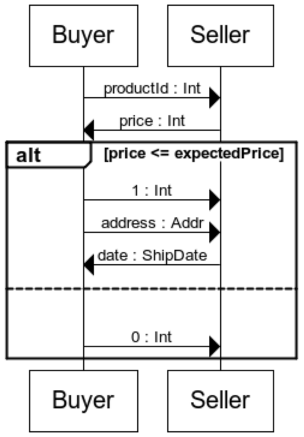

Fig. 1: Sequence diagram for an item purchasing

$$
\begin{array}{lcl}
\texttt{buyer\_ty} & \equiv & \texttt{begin};\,!\texttt{String};\,?\texttt{int};\\
 & & !\{\texttt{ok}:!\texttt{Addr};\,?\texttt{Date};\,\texttt{end},\ \texttt{quit}:\texttt{end}\}
\end{array}
$$

The dual (or complement) of the above session type corresponds to the Seller's communication pattern, namely:

$$
\begin{array}{lcl}
\texttt{seller\_ty} & \equiv & \sim\texttt{buyer\_ty}\\
 & \equiv & \texttt{begin};\,?\texttt{String};\,!\texttt{int};\\
 & & ?\{\texttt{ok}:?\texttt{Addr};\,!\texttt{Date};\,\texttt{end},\ \texttt{quit}:\texttt{end}\}
\end{array}
$$

In the above, $!\texttt{t}$ denotes an output of a value of type `t`, dually for $?\texttt{t}$ which denotes input instead. The type $!\{\texttt{ok} : ...,\ \texttt{quit} : ...\}$ denotes an internal choice (decision based on local values) of the options, while the type $?\{\texttt{ok} : ...,\ \texttt{quit} : ...\}$ denotes an external choice (decision based on received labels) of the options. The options are represented by different labels which are sent/received over the channel. The type `begin` represents the beginning of the conversation, while the type `end` represents the termination of the conversation for a given channel. Traditionally, a program that implements the above protocol uses specialized switch constructs [16] like `outbranch` and `inbranch` to model the internal and external choices respectively:

```
void buyer(buyer_ty c, String p)        | void seller(seller_ty c)
{ send(c, p);                           | { String p = receive(c);
  Double price = receive(c);            |   send(c, getPrice(p));
  Double budget = ...;                  |   inbranch(c) {
  if price <= budget then{              |     case ok : {
    outbranch(c, ok){                   |       Addr a = receive(c);
    Addr a = ...;                       |       ShipDate sd = ...;
    send(c, a);                         |       send(c, sd);
    ShipDate sd = receive(c);           |     }
    }} else outbranch(c, quit){}        |     case quit : { }
}                                       | } }
```

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

$$
\begin{array}{lcl}
\texttt{buyer\_ch} & \equiv & !\texttt{String};\,?\texttt{int};\,((!\texttt{1};\,!\texttt{Addr};\,?\texttt{Date})\vee!\texttt{0})\\
\texttt{seller\_ch} & \equiv & \sim\texttt{buyer\_ch}\\
 & \equiv & ?\texttt{String};\,!\texttt{int};\,((?\texttt{1};\,?\texttt{Addr};\,!\texttt{Date})\vee?\texttt{0})
\end{array}
$$

Superficially, this logical specification looks similar to session type; however, there are several notable differences. Firstly, there is no need for any begin/end declarations since our protocol is expected to be locally captured after creation (without restriction). Secondly, we make use of disjunction[1] instead of some specialized notations for internal and external choices. Thirdly, instead of message labels (such as ok and quit), we may just use values (such as 1 or 0) or even types themselves to capture the distinct scenarios 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. Most importantly, instead of types or values, we allow more general properties (including ghost properties) to be passed into channel to facilitate the verification of functional correctness properties, which can go beyond communication safety. This also includes the use of higher-order channels to model 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.

$$
\begin{array}{lcl}
\texttt{buyer\_chan} & \equiv & !\texttt{String};\,?\texttt{r}{:}\texttt{int}\cdot\texttt{r}{>}0;\,((!\texttt{1};\,!\texttt{Addr};\,?\texttt{Date})\vee!\texttt{0})\\
\texttt{seller\_chan} & \equiv & \sim\texttt{buyer\_chan}
\end{array}
$$

Note that our channel specification uses several abbreviated notations. $?\texttt{1}$ is a short-hand for $?\texttt{r}\cdot\texttt{r}{:}\texttt{int}\wedge\texttt{r}{=}1$, while

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

!String is a shorthand for $!\mathtt{r}\cdot\mathtt{r}{:}\mathtt{String}{\wedge}\mathtt{true}$. The specification seller_chan is the dual specification of buyer_chan. Such dual specification are obtained by inverting the polarity of messages, where input is converted to output and vice-versa. We can also support separation formulae for pointer-based message passing for shared memory implementation. When separation formula is emp we use abbreviated notations, such as $?\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}{>}1$ as a short-hand for $?\mathtt{r}\cdot\mathtt{emp}\wedge\mathtt{r}{:}\mathtt{int}\wedge\mathtt{r}{>}1$. Another issue worth noting is that thread specification and channel specification need not be identical. 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 10 units, as follows:

$$\mathtt{seller\_sp} \quad\equiv\quad ?\mathtt{String}; !\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}{>}10; ((?1; ?\mathtt{Addr}; !\mathtt{Date})\vee ?0)$$

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

```
open(c) with buyer_chan;
(buyer(c, prod) || seller(c));
close(c);

 void buyer(Chan c, String p)          | void seller(Chan c)
    requires C(c, buyer_chan)          |    requires C(c, seller_sp)
    ensures C(c, emp)                  |    ensures C(c, emp)
 { send(c, p);                         | { String p = receive(c);
   Double price = receive(c);          |   send(c, getPrice(p));
   Double budget = ...;                |   int usr_opt = receive(c);
   if (price <= budget) then{          |   if (usr_opt==1){
     send(c, 1);                       |     Addr a = receive(c);
     Addr a = ...;                     |     ShipDate sd = ...;
     send(c, a);                       |     send(c, sd);}
     ShipDate sd = receive(c);         |   else
   } else send(c, 0);                  |     assert usr_opt = 0;
 }                                     |  }
```

The channel is opened in the main process by open which takes as argument the channel specification. One alias of the opened channel with the specification buyer_chan is passed to the thread buyer while the other alias with its dual specification seller_chan is passed to the process seller. The two processes are running in parallel. Each process can have its own separate protocol specification which differs, while being consistent with the channel's specification. The seller process specification seller_sp imposes a stronger property over the sent price, using r>10 instead of r>0 that was captured in the channel specification seller_chan. When a channel is passed into a thread, we will need to ensure that the channel's specification subsume that specified in the thread's specification. For the buyer thread in our example, this means that $\mathcal{C}(\mathtt{c},\mathtt{buyer\_chan}) \vdash \mathcal{C}(\mathtt{c},\mathtt{buyer\_chan})$ which trivially succeeds. For the seller process, we would require $\mathcal{C}(\mathtt{c},\mathtt{seller\_chan}) \vdash \mathcal{C}(\mathtt{c},\mathtt{seller\_sp})$. This second entailment also succeeds because the subsumption for sending operation is contravariant, as illustrated below.

$$\dfrac{\dfrac{\dfrac{\mathtt{r}{>}10 \;\vdash\; \mathtt{r}{>}0}{!\mathtt{r}\cdot\mathtt{r}{>}0 \;\vdash\; !\mathtt{r}\cdot\mathtt{r}{>}10}}{\mathtt{seller\_chan} \;\vdash\; \mathtt{seller\_sp}}}{\mathcal{C}(\mathtt{c},\mathtt{seller\_chan}) \;\vdash\; \mathcal{C}(\mathtt{c},\mathtt{seller\_sp})}$$

Before a channel is used, it must first be opened by open(c) together with an appropriate channel specification.

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

Fig. 2: The Specification Language.

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

Fig. 3: Rules for Dual Specification.

In contrast to previous work (such as [6] where two ends of a single channel are explicitly created, we only use a single channel name but allow aliases, so that complementary operations using send and receive can be communicated over its opened channel. In the end, the main process is allowed to destroy the created channel. Note that the function int getPrice(String) specifies in its postcondition that its result is always greater than 10. With this, the verification of the bodies of both processes succeeds.

III. Session Logic

We develop our session specification language on top of the specification language (in Fig. 2) from [17]. The language allows (user-defined) shape predicates *spred* to specify program properties in a combined domain. Note that such predicates are constructed with disjunctive constraints Φ. A session specification for channel $\mathtt{v}$ 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. A conjunctive abstract program state σ has mainly two parts: the heap (shape) part κ in the separation domain and the pure part π in convex polyhedra domain and bag (multi-set) domain, where π consists of γ, ϕ and φ as aliasing, numerical and multi-set information, respectively. $k^{\mathtt{int}}$ is an integer constant. The square symbols like $\sqsubset$, $\sqsubseteq$, $\sqcup$ and $\sqcap$ are multi-set operators. 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 normalised to the Φ form [17]. The rules to obtain dual specifications are given in Fig. 3.

In our work we make use of the separation logic prover SLEEK [17] to prove whether one formula Δ' in the combined abstract domain entails another one Δ: $\Delta'{\vdash}\Delta{*}\mathtt{R}$. R is called the *frame* which is useful to support sub-structural reasoning rules

$$\frac{\Delta_2 \vdash \Delta_1}{!r\cdot\Delta_1 \vdash !r\cdot\Delta_2} \qquad \frac{\Delta_1 \vdash \Delta_2}{?r\cdot\Delta_1 \vdash ?r\cdot\Delta_2}$$

$$\frac{\begin{array}{c} e_1 \vdash e_2 \\ \mathit{rest}_1 \vdash \mathit{rest}_2 \end{array}}{e_1;\mathit{rest}_1 \vdash e_2;\mathit{rest}_2} \qquad \frac{S_1 \vdash S_2}{\mathcal{C}(c,S_1) \vdash \mathcal{C}(c,S_2)}$$

Fig. 4: Entailment rules for session logic.

of separation logic. We extend the SLEEK rules to support entailment over the session logic formulae (see Fig. 4). The subsumption of the session formulae which correspond to send operations is contravariant while the subsumption of the session formulae corresponding to receiving operations is covariant.

We also need to be able to check the compatibility of two session logic specifications. The rules are given in Fig. 5. The session formula corresponding to sending subsumes the session formula corresponding to receiving. In case of the disjunctions the sending part can have fewer disjunctions than the receiving part. This follows naturally from the behaviour of disjunction during entailment. For simplicity we present the rules for 3 disjunctions on the receiving part and 2 disjunctions on the sending part.

IV. Verification rules

We formalize our approach on a concurrent imperative language enhanced with communication primitives shown in Fig. 6. Our language is an extension of the sequential language from [17]. A program *Prog* written in this language consists of declarations *tdecl*, which can be data type declarations *datat*, predicate definitions *spred* as well as method declarations *meth*. The definitions for *spred* and *mspec* are given in Fig. 2. Our language is expression-oriented, and thus the body of a method (e) is an expression formed by program constructors. The language allows both call-by-value and call-by-reference method parameters. These parameters allow each iterative loop to be directly converted to an equivalent tail-recursive method, where mutations on parameters are made visible to the caller via pass-by-reference. This technique of translating away iterative loops is standard and is helpful in further minimising our core language. The language allows the creation of parallel processes by using the operator $||$. The processes can communicate through channels. A channel is created by `new Chan()` but cannot be used until is has been opened. Each channel is given an alias that can be freely passed. There are two possible kinds of channels, monolithic vs double-ended. Monolithic channel allow an alias to used by multiple parties. Double-ended channel splits a channel into two ends that are to be later used by two parties. Our language uses the more general monolithic channels, but our reasoning system can support either model by simply using a different set of specifications for double-ended channels. We use the same verification rules as in HIP/SLEEK , but for the processes and the channel operations we provide specifications in term of pre and post-conditions.

$$\begin{array}{lll}
\mathit{Prog} & ::= \mathit{tdecl}^*\ \mathit{meth}^* & \mathit{tdecl} ::= \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} & ::= \mathtt{data}\ c\ \{\ (t\ v)^*\ \} & \\
t & ::= c \mid \mathit{prim} \mid \mathit{Chan} \mid \mathit{dyn} & \mathit{prim} ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{void} \\
\mathit{meth} & ::= t\ \mathit{mn}\ (\mathit{ref}\ (t\ v)^*;(t\ x)^*)\ \mathit{mspec}\ \{e\} & \\
e & ::= \mathit{null} \mid k^{\mathit{prim}} \mid v \mid v.f \mid v := e \mid v_1.f := v_2 \mid e_1;e_2 & \\
 & \mid \mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{t}\ v;\ e \mid mn(v^*;x^*) & \\
 & \mid \mathtt{new}\ c(v^*) \mid \mathtt{new\ Chan}() \mid e_1 || e_2 & \\
 & \mid \mathtt{open}(c)\ \mathtt{with}\ \mathit{spred} \mid \mathtt{close}(c) \mid \mathtt{send}(c,v) \mid \mathtt{receive}(c) &
\end{array}$$

Fig. 6: A Concurrent Imperative Language with Sessions.

A channel can be opened by `open` with some channel specification S. After opening we have two aliases of the same channel, one having the specification S and the other one having the complimentary specification $\sim S$ as follows:

```
void open(Chan c) with S
    requires emp
    ensures C(c,S) * C(c,∼S)
```

A channel can be closed (or destroyed) only when both aliases are available and both have consumed their specifications, as follows:

```
void close(Chan c)
    requires C(c,emp) * C(c,emp)
    ensures emp
```

In general for a correct usage of a channel we must have the following situation:

```
open(c) with S1
//C(c,S1) * C(c,∼S1)
//splitting into C(c,S1) || C(c,∼S1)
Process1(c)  ||  Process2(c)
//both processes consume their channels
//C(c,emp) * C(c,emp)
close(c)
```

In contrast to session types, we need only rely on two communication operations over a channel: `send` and `receive`. The specifications of the operations are given below. Note that `res` is a reserved word denoting the result returned by `receive` while `L(x)` is a session logical formula about `x`.

```
t receive(Chan c)
    requires C(c,?r:t·L(r);rest)
    ensures L(res) * C(c,rest)

void send(Chan c, t x)
    requires C(c,!x:t·L(x);rest) * L(x)
    ensures C(c,rest)
```

In a 2-party session, one channel is typically sufficient for communication between the two parties. Let us denote the two parties by two processes `P(c)` and `Q(c)`, where `c` is the communication channel. Apart from the communication channel specification we can also have a communication specification for each party, `P_sp` and `Q_sp`. In general, the specifications of the processes can be written as follows:

```
t P(Chan c)
    requires C(c,P_sp) * Pre1
    ensures C(c,R1) * Post1

t Q(Chan c)
    requires C(c,Q_sp) * Pre2
    ensures C(c,R2) * Post2
```

$$\frac{\begin{array}{c}\textit{compatible}(!r\cdot\Delta_1;P_1,?r\cdot\Delta'_1;Q_1)\\ \textit{compatible}(!r\cdot\Delta_2;P_2,?r\cdot\Delta'_2;Q_2)\\ \textit{disjoint}((?r\cdot\Delta'_1;Q_1),(?r\cdot\Delta'_2;Q_2),(?r\cdot\Delta'_3;Q_3)))\end{array}}{\textit{comp\_aux}(((!r\cdot\Delta_1;P_1)\vee(!r\cdot\Delta_2;P_2)),((?r\cdot\Delta'_1;Q_1)\vee(?r\cdot\Delta'_2;Q_2)\vee(?r\cdot\Delta'_3;Q_3)))}$$

$$\frac{\Delta_1\vdash\Delta_2}{\textit{comp\_aux}(!r\cdot\Delta_1,?r\cdot\Delta_2)}\quad\frac{\textit{comp\_aux}(P_2,P_1)}{\textit{comp\_aux}(P_1,P_2)}\quad\frac{\textit{comp\_aux}(h_1,h_2)\quad\textit{compE}(\textit{rest}_1,\textit{rest}_2)}{\textit{compE}(h_1;\textit{rest}_1,h_2;\textit{rest}_2)}$$

$$\frac{\textit{compE}(L,R)\vee\textit{compL}(L,R)\vee\textit{compR}(L,R)}{\textit{compatible}(L,R)}\quad\frac{\textit{compE}(L,R)\vee\textit{compL}(L,R)}{\textit{compL}(?r\cdot\Delta;L,R)}\quad\frac{\textit{compE}(L,R)\vee\textit{compR}(L,R)}{\textit{compR}(L,?r\cdot\Delta;R)}$$

Fig. 5: Session Specifications Compatibility rules.

Operation `close` must ensure that the communication has been completed and it is empty. In the following example `close` fails since the communication is not empty. The example uses a recursive session specification S_2.

```
S₂ ≡ !String; S₂

open(c) with S₂;
//𝒞(c, S₂) ∗ 𝒞(c, ∼S₂)
//𝒞(c, S₂)          || //𝒞(c, ∼S₂)
for(i = 1 to 5)     || for(i = 1 to 10)
  send(c, i);       ||   int x = receive(c);
//𝒞(c, S₂) ∗ 𝒞(c, ∼S₂)
close(c);//FAILS!
```

The channel can be dynamically typed. Dynamic types in our langauge are denoted by `Dyn`. For instance the type signature of `send` and `receive` are essentially dynamically typed:

```
void send(Chan c, Dyn val){...}
Dyn receive(Chan c){...}

send(c, 3); send(c, "...");
int r = (int) receive(c);
String r = (String) receive(c);
```

Our automated verification rules help guarantee communication safety via type-safe casting. We can support dynamic type values by using a specialized `switch` construct, as follows:

```
Dyn t = receive(c);
switch t with {
  v1 : int → ...
  v2 : String → ...
}
```

Alternatively, we may also support it via type testing with conditional constructs, as follows:

```
Dyn t = receive(c)
if (type(t) = int) {v1 = (int)t; ...}
else if (type(t) = String) {v2 = (String)t; ...}
else {assert false;}
```

Using dynamic testing of types a recursive channel specification can be written as:

```
S₃ ≡ !Object; (S₃ ∨ !0)
```

However, using only type-safe casting without run-time type testing, our channel specification would have to be written as follows where each disjunct starts with the same type:

```
S₄ ≡ !Object; (!1; S₄ ∨ !0)
```

V. Higher-order Session Logic

Delegation is one of the most important distinctions between session types and other communication calculus based methods. Without the enforcement of delegation we cannot speak about session types verification. To attest our theorie's correctness we consider the following example, taken from [10]. From global viewpoint the Buyer sends a product identifier to the Seller. The Seller provides the Buyer a price for the requested product. The Buyer can decide, according to their internal rules, to accept the product or not. If the Buyer accepts 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 final 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. Before starting the development of such a system, it is recommended to clearly specify the collaboration between the entities with a sequence diagram as it is in Fig. 7. This diagram can be considered as a graphical specification of our session logic. The transformation can be done automatically.

The projection of this problem's protocols is given below:

```
buy_sp   ≡  !int; ?r:Double · sprice(r);
            ((!1; !Addr; ?Date) ∨ !0)
ship_sp  ≡  (?1; ?Prod; ?r:Chan · 𝒞(r, ?Addr; !Date);
            !r:Chan · 𝒞(r, emp)) ∨ ?0
```

where `sprice` is defined, as below:

```
sprice(root) ≡ ∃i, f : root↦Double⟨i, f⟩
                       ∧i ≥ 0 ∧ f ≥ 0 ∧ i + f > 0
```

The first specification of the protocol `buy_sp` describes the communication from the Buyer's point of view. The protocol does not take into consideration that it could be delegated toward the Shipper. From Buyer point of view the delegation is transparent, but for the Seller the delegation is explicit and thus included in the protocol specification `ship_sp`. This specification formulates the protocol from the Shipper perspective and highlights the process of the delegation. The delegation is expected to happen after receiving the Buyer's decision and the product details. The Buyer's choice is transmitted

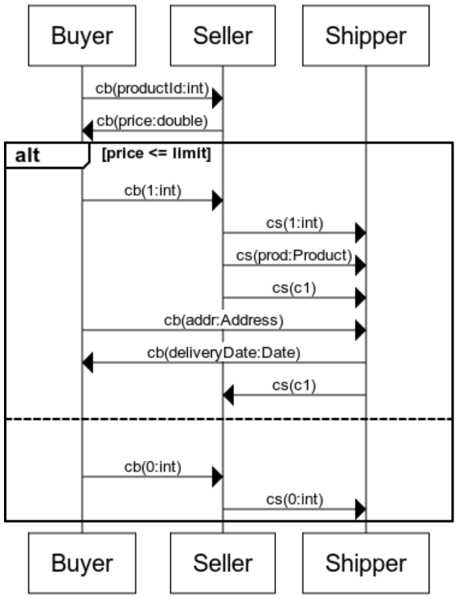

Fig. 7: Sequence diagram of shipping delegation

to the Shipper to help it to consume the protocol. This process is required because of the early synchronization of the processes. If the latter synchronization of the processes would be permitted, allowing the parts to make connections on demand, then the deadlock freedom of the communication process would be hardly provable.

From delegation perspective the specification is precise and highlights the state of the transmitted channel, which must be insured by the sender and can be assumed by the receiver.

The program codes which can implement the above protocol are given below:

```
open(cb) with buy_sp;
open(cs) with ship_sp;
int prod = getProdID();
Addr a = getAddr();
Double budget = getBudget();

(buyer(cb, prod, budget, a) ||
seller(cb, cs) || shipper(cs));

close(cb);
close(cs);
```

```
shipper(Chan c)
    requires 𝒞(c, ship_sp)
    ensures  𝒞(c, emp)
{int usr_opt = receive(c);
if (usr_opt == 1) {
   Prod p  = receive(c);
   Chan cd  = receive(c);
   Addr a  = receive(cd);
   Date sd  = cShip(a, p);
   send(cd, sd);
   send(c, cd);
} else if(usr_opt == 0) {
} else assert false; }
```

```
seller(Chan cb, Chan cs)
 requires 𝒞(cb, ∼buy_sp)
      ∗ 𝒞(cs, ∼ship_sp)
 ensures 𝒞(cb, emp) ∗ 𝒞(cs, emp)
{int id  =  receive(cb);
 send(cb, getPrice(id));
 int usr_opt = receive(cb);
 if(usr_opt == 1) {
  send(cs, 1);
  Prod p  =  getProd(id);
  send(cs, p);
  send(cs, cb);
  cb = receive(cs);
 } else if(usr_opt == 0)
  send(cs, 0);
 else assert false; }
```

```
buyer(Chan c, int id,
 Double budget, Addr a)
   requires 𝒞(c, buy_sp)
   ensures   𝒞(c, emp)
{send(c, id);
 Double price  =  receive(c);
 if(price  <=  budget) {
  send(c, 1);
  send(c, a);
  Date sd  =  receive(c);
 } else send(c, 0); }
```

Considering the previous code as a correct implementation of our example then the verification consists of proving that the involved entities are playing their role according to the previously mentioned specifications. The most complex role of our protocol, which results in a complex implementation (see *seller* function) is the seller's role. The verification process consists of applying out the session logic specifications and the entailment rules. Some of the essential symbolic states of seller's function verification can be found in the following annotated source code:

```
// 𝒞(cb,  ?int; !r:Double · sprice(r); ((?1; ?Addr; !Date)  ∨  ?0)) ∗
// 𝒞(cs,  (!1; !Prod; !r:Chan · 𝒞(r,  ?Addr; !Date); ?r:Chan ·
// 𝒞(r, emp))  ∨  !0)
1     int id  =  receive(cb);
// 𝒞(cb,  !r:Double · sprice(r); ((?1; ?Addr; !Date)  ∨  ?0)) ∗
// 𝒞(cs,  (!1; !Prod; !r:Chan · 𝒞(r,  ?Addr; !Date); ?r:Chan ·
// 𝒞(r, emp))  ∨  !0)
2     send(cb, getPrice(id));
3     int usr_opt = receive(cb);
4     if (usr_opt == 1) {
// 𝒞(cb,  ?Addr; !Date)  ∗  𝒞(cs,  (!1; !Prod; !r:Chan ·
// 𝒞(r,  ?Addr; !Date); ?r:Chan · 𝒞(r, emp))  ∨  !0)
5         send(cs, 1);
6         Prod p  =  getProd(id);
7         send(cs, p);
// 𝒞(cb,  ?Addr; !Date)  ∗  𝒞(cs,  !r:Chan · 𝒞(r,  ?Addr;
// !Date); ?r:Chan · 𝒞(r, emp))  ∗  p↦Prod⟨_⟩
8         send(cs, cb);
// 𝒞(cs,  ?r:Chan · 𝒞(r, emp))  ∗  p↦Prod⟨_⟩
9         cb = receive(cs);
// 𝒞(cs, emp) ∗ 𝒞(cb, emp)  ∗  p↦Prod⟨_⟩
10    } else if (usr_opt == 0) {
// 𝒞(cb, emp)  ∗  𝒞(cs,  (!1; !Prod; !r:Chan · 𝒞(r,  ?Addr;
// !Date); ?r:Chan · 𝒞(r, emp))  ∨  !0)
11        send(cs, 0);
// 𝒞(cb, emp)  ∗  𝒞(cs, emp)
12    } else assert false;
```

Given the forward verification style of our approach, the symbolic execution of the `seller` method starts by assuming its precondition, namely $\mathcal{C}(\texttt{cb}, \sim\texttt{buy\_sp}) * \mathcal{C}(\texttt{cs}, \sim\texttt{ship\_sp})$, where $\sim\texttt{buy\_sp}$ and $\sim\texttt{ship\_sp}$ are computed using the duality rules from Fig. 3. Next, after the symbolic execution of line 1, the program's symbolic state reflects the receiving of an `int` value over channel `cb` by removing `?int` from the specification of `cb`. Formally, if the precondition of `receive` holds, the next symbolic state comprises the frame resulted after proving the precondition along with the method's post-condition. The same rationale applies for lines 2-3 and 5-8, with the remark that line 3 is represented as a choice in the specification of `cb`: $((?1; ...) \vee ?0)$, while line 8 denotes a delegation. It is important to note that our approach is composable, e.g. the verification of the `send` method follows the same rule whether it is used for delegation or merely for sending an int value over a specified channel.

VI. Full Expressivity of Separation Logic

We demonstrate the expressiveness of our approach by using a very simple 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. 8

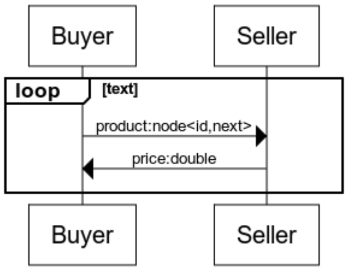

Fig. 8: Sequence diagram for items purchasing

Given the following data node declaration:

```
data node{int id; node next; }
```

a separation logic based specification language conveniently provides support in writing the necessary heap predicates, such as the linked list definition below, which could then be used by the buyer to send the list of product identifiers:

$$\texttt{pred ll(root)} \quad \equiv \quad \begin{array}{l}\texttt{root} = \texttt{null} \vee \\ \exists\, \texttt{q} \cdot \texttt{root}{\mapsto}\texttt{node}\langle\_, \texttt{q}\rangle * \texttt{ll(q)}\end{array}$$

The above predicate can be used in disjunctions over the protocol. However, in order to avoid ambiguous protocols we must impose disjoint conditions. For instance, the following channel is ambiguous:

$$?\texttt{x:node} \cdot \texttt{ll(x)} \;\vee\; ?\texttt{x:node} \cdot \texttt{x} = \texttt{null}$$

while the following is unambiguous:

$$(?\texttt{x:node} \cdot \texttt{ll(x)} \;\wedge\; \texttt{x} \neq \texttt{null}) \;\vee\; (?\texttt{x:node} \cdot \texttt{x} = \texttt{null})$$

Though its condition contains ghost values, the receiver side can still exploit it.

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

$$\texttt{buy\_lsp} \equiv \begin{array}{l} !\texttt{p :node} \cdot \texttt{p} = \texttt{null};\ \vee \\ !\texttt{p :node} \cdot\ \texttt{p}{\mapsto}\texttt{node}\langle\_,\_\rangle; \\ ?\texttt{price :Double} \cdot \texttt{sprice(price)}; \texttt{buy\_lsp}\end{array}$$

The protocol specification asserts that each outward transmission of a not null node must be followed by an input of type Double. In other words, the party which obeys the buy_lsp protocol, the Buyer in this case, first transmits an element whose details are stored in the node structure, and then expects a response from the Seller in the form of a price for the considered element. The communication terminates once the Buyer has received a null reference from the seller, which marks the end of the list (the list which stores the elements of interest). The communication termination condition is emphasized in the first disjunct of the protocol specification: where p must be null, as opposed to the case where the list still contains elements signaled by $\texttt{p}{\mapsto}\texttt{node}\langle\_,\_\rangle$ and thus the specification continues to iterate over buy_lsp.

The program code implementing the entry point of the above protocol is given below:

```
...
open(c) with buy_lsp;
(buyer(c) || seller(c));
close(c);
```

The code for the two processes running in parallel, buyer and seller, respectively, is as follows:

```
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){
    Double p = price(it.id);
    send(c, p);
    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);}
}
```

Consistent with our previous examples, the verification of buyer starts by assuming its precondition, namely $\mathcal{C}(\texttt{c}, \texttt{buy\_lsp})$. The abstract states following the symbolic execution of buyer are highlighted below:

```
// C(c, buy_lsp)
node it = getItems();
// C(c, buy_lsp) * ll(it)
recvPrices(c, it);
// C(c, emp)
```

The list used to store the products is transmitted to the Seller on a node by node basis in the recvPrices method. The fact that the communication uses node transmission serves a double scope: for sharing product information in a compound manner 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 can be done via the transmitted data, without the need of a different synchronization package. Generally, the session type techniques require a flag transmission at each iteration in order to ensure that the loops have same iterations. In contrast, our work allows the verification of a more optimal implementation. More precisely, in our work the communication entities rely on the transmitted data, offering support for a more expressive and precise specification courtesy of [17] and the benefits of separation logic.

By using separation logic to specify and verify this example it is ensured that if the verification succeeds, the communication terminates and respects its protocol. This guarantee is justified through the usage of separating conjunction, $*$, in the definition of ll which implicitly states that each element in the list resides in disjoint heap. In other words, it guarantees that iterating over a linked list such as ll doesn't produce non-productive loops. For brevity of the presentation, we stress on the assumption that the data structures are immutable (i.e. the lists are not updated during the communication), the physical communication channels suffer no loss of transmitted information and that the seller is able to provide a price for each product.

We next present a detailed view of the abstract program states for the symbolic execution of `recvPrices`:

```
1 void recvPrices(Chan c, node it)
2   requires 𝒞(c, buy_lsp) ∗ ll(it)
3   ensures  𝒞(c, emp)
4 {
// 𝒞(c, buy_lsp)  ∗  ll(it)
5   if(it! = null){
// 𝒞(c, buy_lsp)  ∗  ll(it) ∧ it ≠ null
6     node nxt = it.next;
// 𝒞(c, buy_lsp)
//      ∗  it↦node⟨p₀, q⟩  ∗  ll(q) ∧ it ≠ null ∧ nxt = q
7     int id = it.id;
// 𝒞(c, buy_lsp)
//     ∗  it↦node⟨p₀, q⟩  ∗  ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p₀
// − − − − − − − − − − − − − − − − − − − − − − − − − −
// 𝒞(c, !p :node · p = null∨!p :node · p↦node⟨_, _⟩;
//    ?price :Double · sprice(price); buy_lsp)
//     ∗  it↦node⟨p₀, q⟩  ∗  ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p₀
8     send(c, it);
// 𝒞(c, ?price :Double · sprice(price); buy_lsp)
//     ∗  ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p₀
9     Double price = receive(c);
// 𝒞(c, buy_lsp)  ∗  sprice(price)
//     ∗  ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p₀
10     procPrice(id, price);
// 𝒞(c, buy_lsp)  ∗  sprice(price)
//     ∗  ll(q) ∧ it ≠ null ∧ nxt = q ∧ id = p₀
11     recvPrices(c, nxt);
// 𝒞(c, emp)  ∗  sprice(price) ∧ it ≠ null ∧ nxt = q ∧ id = p₀
12  } else {
// 𝒞(c, buy_lsp) ∧ it = null
// − − − − − − − − − − − −
// 𝒞(c, !p :node · p = null∨!p :node · p↦node⟨_, _⟩;
//    ?price :Double · sprice(price); buy_lsp) ∧ it = null
13     send(c, it);
// 𝒞(c, emp) ∧ it = null
14  }
// 𝒞(c, emp) ∧ it ≠ null ∨ 𝒞(c, emp) ∧ it = null
// − − − − − − − − − − − − − − − − − − − − − −
// 𝒞(c, emp)
15 }
```

The verification of the `recvPrices` method follows closely the same rationale as the examples detailed in the previous sections. Special attention, however, is given to the verification of the `send` call at line 8, which emphasizes the copyless character of the underlying communication. More precisely, `send` consumes the `node` pointed by `it` from the Buyer's abstract state (while its receiving counterpart at the Seller's side inherits the permission to access the transmitted `node`). Moreover, the Buyer's program state correctly reflects the progression of the communication protocol by consuming the output !p :`node` from the session specification. It is important to note that our entailment technique safely prunes the contradicting disjuncts while unfolding the specialized session predicate. The contradiction usually refers to the conflict between the conditions of the session choice and the program's state guard. Specifically, our calculus [18] prunes the session disjunct which refers to the communication termination, !p :`node` · `p` = `null`, provided that the calling context is guarded by `it` ≠ `null`, the recursion condition.

VII. Deadlock Detection

In this section we discuss how our approach detects deadlocks during two-party channel asynchronous communication. Our deadlock detection is based on the assumption that `send` method is non-blocking, while `receive` may be blocked.

Consider a single channel communication between two parties. Let us analyse the possible situations that may lead to a deadlock. After opening, each channel has two aliases: one with the given specification `Z` and the other one with its dual specification ∼`Z`. The two aliases of a single channel are used by two parties of the channel. In the main procedure, they are passed to the two parties (processes) that are involved in communications. Intra-channel deadlock may occur when the 2-parties of the channel are not synchronized with an excess of blocking `receive` methods, before `send` methods. That means both parties have different consumption of the resource 𝒞(`c`, `Z`) and 𝒞(`c`, ∼`Z`). In order to detect the deadlock we have to do a synchronization check when the parties' processes are merged (or joined) together. The check consists of a compatibility verification of the remaining resources from the same channel, say 𝒞(`c`, `Z1`) and 𝒞(`c`, `Z2`). The rules are given in Fig. 5.

Let us consider a simple example where `Z` =?`String`. The example shows the situation when at the merging point one party has the session specification 𝒞(`c`, `emp`) and the other has the specification 𝒞(`c`, !`String`). These two channel specifications are incompatible, with an outstanding `send`, leading to an intra-channel deadlock from the blocking `P(c)` method.

```
open(c) with ?String
// 𝒞(c, ?String) ║ // 𝒞(c, !String)
       P(c)       ║        skip
// 𝒞(c, emp)  ∗  𝒞(c, !String)
//synchronization check FAILS
Q(c)
```

Let us now swap the positions of the `P(c)` and `Q(c)` methods, such that non-blocking *send* method in `Q` is executed first. In this case, our channel specifications 𝒞(`c`, ?`String`) and 𝒞(`c`, `emp`) remains compatible since it has an excess of `receive` operation that could always be invoked at a later time. The compatibility check succeeds on its merge point, followed by `P(c)` method call which would then execute its `receive` command. There is therefore no intra-channel deadlock here.

```
open(c) with ?String
// 𝒞(c, !String) ║ // 𝒞(c, ?String)
       Q(c)       ║        skip
// 𝒞(c, emp)  ∗  𝒞(c, ?String)
//synchronization OK
P(c)
// 𝒞(c, emp)  ∗  𝒞(c, emp)
```

A more general scenario is outlined next for some arbitrary `P(c)` and `Q(c)` method. We simply need to apply compatibility checking at the merge point to check if intra-channel deadlock has already occurred at the point.

```
open(c) with Z
// C(c,Z)  ‖ // C(c,~Z)
   P(c)    ‖     Q(c)
// C(c,K1) * C(c,K2)
//synchronization check
```

VIII. Related Work

Session types have been studied in many contexts, including functional languages (Haskell, F#, multi-threaded ML), Java, mobile processes, higher-order processes, operating systems and web services [9], [19], [10], [20]. Most of these works aim to prove the correctness of the communications and only partially address the problem of deadlock detection. We started our work by analyzing session types for object-oriented languages [10] which proposes an incomplete solution for deadlock detection. There are several works [21], [22], [23], [24], [25], [26] which enforce the session types specification by dynamic verification. These works address the dynamic verification of the protocol specifications but their verification is not exhaustive and can not be applied on an early stage of development.

The relation between the π-calculus and separation logic is studied in [27] and [28], but their work gives a treatment of the π-calculus based on the semantic theory of separation logic, without concentrating on protocol verification. The same idea was also studied in relation to Hoare logic in [29]. From the perspective of other protocol specification languages, there is a work [30] which tries to encode the CSP into Hoare Logic, but they have encoded only the send and receive commands without the branching. [14] suggests a logic to extend multiparty session specifications, by enriching the assertion language studied in [13] with capability to refer to virtual states local to each network principal. Lately, Villard et al. [15] have been developing a logic relying on state-based global contracts, while our more general logic of session is built as an extension of separation logic with disjunction to support communication choices.

IX. Conclusion

We propose a novel session logic with disjunctions to specify and verify the implementation of the communication protocols. Our current logic is based on only two-party channel sessions, but it is capable to naturally handle delegation through the use of higher-order channels. Due to our use of disjunctions to model both internal and external choices, we can use only conditional statements to support such choices, as opposed to specialized switch constructs in prior proposals. As our proposal is based on an extension of separation logic, we can support heap-manipulating program and copyless message passing. Our session logic is being implemented on top of the Hip/Sleek system [17]. In future we plan to extend our session logic to multi-party and multi-channel specifications.

Acknowledgments

We like to thank Prof. Wei-Ngan Chin for his invaluable advices and fruitful discussions. We also like to thank Dr. Shengchao Qin for fruitful discussions. This work is supported by Siemens grant no. 7472/3202246933 and by MoE Tier-1 NUS research grant R-252-000-525-112.

References

[1] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund, "Designing safe, reliable systems using scade," in *Leveraging Applications of Formal Methods*. Springer, 2006, pp. 115–129.

[2] S. Schneider, *The B-method: An introduction*. Palgrave Oxford, 2001.

[3] W.-N. Chin, C. David, and C. Gherghina, "A HIP and SLEEK verification system," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 9–10.

[4] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[5] R. Milner, "A Calculus of Communication Systems. LNCS, vol. 92," 1980.

[6] J. Villard, É. Lozes, and C. Calcagno, "Proving copyless message passing," in *Programming Languages and Systems*. Springer, 2009, pp. 194–209.

[7] R. Pucella and J. A. Tov, "Haskell Session Types with (Almost) No Class," *SIGPLAN Not.*, vol. 44, no. 2, Sep. 2008.

[8] R. DeLine and M. Fähndrich, "Enforcing high-level protocols in low-level software," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 59–69, 2001.

[9] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira, "Modular session types for distributed object-oriented programming," in *ACM Sigplan Notices*, vol. 45, no. 1. ACM, 2010, pp. 299–312.

[10] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou, "Session types for object-oriented languages," in *ECOOP 2006–Object-Oriented Programming*. Springer, 2006, pp. 328–352.

[11] S. Gay and M. Hole, "Subtyping for session types in the π calculus," *Acta Informatica*, vol. 42, no. 2-3, pp. 191–225, 2005.

[12] D. Mostrous and N. Yoshida, "Session typing and asynchronous subtyping for the higher-order π-calculus," *Information and Computation*, vol. 241, pp. 227–263, 2015.

[13] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida, "A theory of design-by-contract for distributed multiparty interactions," in *CONCUR 2010-Concurrency Theory*. Springer, 2010, pp. 162–176.

[14] L. Bocchi, R. Demangeon, and N. Yoshida, "A multiparty multi-session logic," in *Trustworthy Global Computing*. Springer, 2013, pp. 97–111.

[15] É. Lozes and J. Villard, "Shared contract-obedient channels," *Science of Computer Programming*, vol. 100, pp. 28–60, 2015.

[16] R. Hu, N. Yoshida, and K. Honda, "Session-Based Dsitributed Programming in Java," in *ECOOP 2008 Object-Oriented Programming*, ser. Lecture Notes in Computer Science. Springer, 2010, pp. 516–541.

[17] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated Verification of Shape, Size and Bag Properties Via User-Defined Predicates in Separation Logic," *Sci. of Comp. Prog.*, vol. 77, pp. 1006–1036, 2012.

[18] W.-N. Chin, C. Gherghina, R. Voicu, Q. L. Le, F. Craciun, and S. Qin, "A specialization calculus for pruning disjunctive predicates to support verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11, 2011, pp. 293–309.

[19] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda, "Type-Safe Eventful Sessions in Java," in *ECOOP 2010 Object-Oriented Programming*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 329–353.

[20] M. Dezani-Ciancaglini, E. Giachino, S. Drossopoulou, and N. Yoshida, "Bounded Session Types for Object Oriented Languages," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, 2007, vol. 4709, pp. 207–245.

[21] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini, "Information flow safety in multiparty sessions," *Mathematical Structures in Computer Science*, pp. 1–43, 2011.

[22] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, and N. Yoshida, "Asynchronous distributed monitoring for multiparty session enforcement," in *Trustworthy Global Computing*. Springer, 2012, pp. 25–45.

[23] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida, "Monitoring Networks through Multiparty Session Types," in *FMOODS/FORTE 2013*, ser. LNCS, vol. 7892. Springer, 2013, pp. 50–65.

[24] E. R. B. Marques, F. Martins, V. T. Vasconcelos, N. Ng, and N. D. Martins, "Towards deductive verification of MPI programs against session types," in *PLACES'13*, ser. EPTCS, vol. 137. Open Publishing Association, 2013, pp. 103–113.

[25] R. Neykova, N. Yoshida, and R. Hu, "Local Verification of Global Protocols," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Springer Berlin Heidelberg, 2013, pp. 358–363.

[26] K. Honda, R. Hu, R. Neykova, T.-C. Chen, R. Demangeon, P.-M. Denilou, and N. Yoshida, "Structuring Communication with Session Types," in *Concurrent Objects and Beyond*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, vol. 8665, pp. 105–127.

[27] A. Turon and M. Wand, "A Resource Analysis of the π-calculus," *Electronic Notes in Theoretical Computer Science*, vol. 276, pp. 313 – 334, 2011.

[28] T. Hoare and P. O'Hearn, "Separation Logic Semantics for Communicating Processes," *Electronic Notes in Theoretical Computer Science*, vol. 212, pp. 3 – 25, 2008, proceedings of the First International Conference on Foundations of Informatics, Computing and Software (FICS 2008).

[29] T. F. Melham, "A Mechanized Theory of the pi-calculus in HOL ," in *Nordic Journal of Computing*, 1992.

[30] L. Lamport and F. B. Schneider, "The "Hoare Logic" of CSP, and All That," *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 2, Apr. 1984.

Appendix

The specifications of the auxiliary functions from sections VI and V are given below:

```
node getItems()
    requires emp
    ensures  ll(ret)

void freeNode(node it)
    requires it↦node⟨_, _⟩
    ensures emp ∧ it = null
```

```
void procPrice(int id,
        Double price)
    requires sprice(price)
    ensures emp
```

Fig. 9: Auxiliary functions from section VI

The previously used data structures are defined below:

```
data Double{
    int i;//integer part
    int f;//fractional part
}

data String{
    int strid;
}

data Prod{
    int id;
}
```

```
data Date{
    int year;
    int month;
    int month;
}

data Addr{
    int nr;
    String city;
}
```

Fig. 10: Data structures

```
int getProdID()
    requires emp
    ensures  res > 0

Addr getAddr()
    requires emp
    ensures res↦Addr⟨_, _⟩

Prod getProd(int id)
    requires id > 0
    ensures res↦Prod⟨id⟩
```

```
Double getBudget()
    requires emp
    ensures res↦Double⟨i, f⟩
            ∧  ≥ ∧f ≥ 0

Date cShip(Addr a, Prod p)
    requires a↦Addr⟨_, _⟩
           ∧ p↦Prod⟨id⟩ ∧ id > 0
    ensures res↦Date⟨_, _, _⟩

Double getPrice(int id)
    requires id > 0
    ensures sprice(res)
```

Fig. 11: Auxiliary functions from section V