Session Logic and its Applications in Railway Industry

Kiss Tibor

Supervisor: Prof. Dr. Bazil Pârv

Department of Mathematics and Computer Science
Babeş-Bolyai University Cluj-Napoca
Kogălniceanu str. 1, RO-400084

Doctor of Philosophy

Computer Science November 2016

Table of contents

Chapter 1
Introduction

In today's world, we are becoming increasingly dependent on equipment that is controlled by software, and it seems that in the near future it will become an ever more indispensable part of our day-to-day routines. Such equipment is prevalent in so many important areas of our lives that we can hardly get through the day without it. Consequently, it is crucial for such equipment to be dependable, especially in critical environments such as hospitals, aeronautics, the automotive and railway industries etc.

Software programs which control such equipment are now widely used and play a very important role. Therefore, low-quality software is simply not acceptable.

In order to provide the high-quality software needed to meet such demanding standards, a great effort has been put into software verification and validation, similar to other engineering disciplines. Despite this effort, the most commonly used method of increasing the quality of industrial programs is that of testing. However, it is not an adequate method of excluding errors from a program ("Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence" [37]). It is no wonder then, that even systems which have been extensively tested can fail, sometimes leading to financial or, in the worst case, loss of human life. [129, 7]. To address this potential failure, many tools based on formal methods have emerged to support the automatic or semi-automatic verification of industrial programs. [1, 108, 26, 2, 30]. Over the past few years, some of these tools have registered a significant increase in their quality [18, 6], and are therefore used with some success in the industry. Despite their maturity, these tools do have some shortcomings. Firstly, they are designed as a means of developing software from scratch, rather than as a support for assisting the maintenance of existing safety-critical programs, written in general purpose imperative languages. Moreover, they are developed to model programs which are executed sequentially, without support in designing concurrent asynchronous software.

More precisely, these tools do not target the verification of wide-area distributed systems software, which have become a significant part of modern programming, and in general, are still written within traditional programming languages in the industry. Furthermore, distribution problems are often modelled as informal specifications and design patterns, with minimal or no support for formal verification. The problem becomes even more serious if we consider that software products which control large safety critical infrastructures (for example: electricity and gas distribution systems, telecommunication infrastructures and systems that control and monitor railway interlocking systems) have been mainly developed in simple imperative languages, built for sequential architectures without mechanisms to exhaustively verify their safety properties.

Considering all of the above problems, we can conclude that providing a useful formal method for the verification of railway interlocking systems for our partner - (Railway Automation Division of Siemens) - is very challenging, and even more so if we consider the verification of geographic railway interlocking systems. These systems, which in general must be concurrent, distributed and be able to control large infrastructures (for example the railway infrastructure of a country), are controlled by programs which are not directly targeted by the previously mentioned industrial tools. The formal verification of these programs hides some unexpected challenges which are not immediately obvious: Firstly, it is very difficult to specify global constraints in such huge systems. Secondly, assuming that these previously mentioned constraints are specified, it would be very challenging to verify them because of the abstract and incomplete nature of the software specification.

Analysing current industrial practices, along with the state of the art programming languages and tools for developing distributed railway interlocking software, we have concluded the following: Without a means of routinely and reliably building concurrent and distributed systems - following some correct-by-construction principles - the technical progress of these systems will cease. Furthermore, the price of such systems - considering the development and maintenance costs - remains high [47]. On the other hand, trying to follow the functional development principle of railway interlocking software [14, 5] and to explore the state space of such a system seems also to be generally impossible [5]. Verification generally leads to state explosion, therefore a new methodology of verifying interlocking software precision would be highly welcome.

Considering the above problems of formal verification of distributed system software which comprises the verification of geographic railway interlocking [1], and with the success of tools like [1], [108] and [26] for the development and verification of non-concurrent software, there is now a deeper desire for similar tools for concurrent software. So, the communication requirement - ubiquitous in software systems for information exchange between software entities and the communication of these systems with their environments - must be properly verified to affirm the system's precision.

[1]We consider geographic railway interlocking as particular case of the previously mentioned distributed systems

1.1 Goals and Hypothesis

The general aim of this thesis is to contribute to the scientific foundations of the verification of safety-critical concurrent programs. Since the safety of a distributed system depends upon the correct execution of its concurrent program, our particular goal is to provide an automatic verification technique which is based on a theoretically well-founded formal method for supporting the verification of asynchronous concurrent programs. An additional goal of this thesis is to provide a set of case studies and development methods based on the previous formalism, to encourage the application of this technique in the railway industry, especially in the verification of geographic interlocking and train control systems.

Due to its importance, a number of researchers have, over the past few decades, focused on the problems of ensuring safe communication.

CSP (Communicating Sequential Processes) [21] and CCS (Calculus of Communicating Systems) [90] 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 [117]. In the last decade, session types have been integrated into a number of programming languages and process calculi, including functional languages [105, 33], object-oriented languages [52, 36], calculi of mobile processes [51], and higher-order processes [96]. 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 that one party must ensure, and upon which the other party can rely. There was also a proposal for multi-party session logic [12], but this logic tries to also summarise the effects of processes involved in the protocol.

Although, these theories are very promising, their results are currently impracticable in the industry for several reasons: Firstly, these verifications require a syntactic correspondence between primitives of the programming language and primitives of the protocol specification language. Additionally, the majority of these mechanisms also require some restrictions on reference aliasing, to enable a precise tracking of channel endpoints.

Therefore, our **hypothesis** is that there exists an extension of separation logic which allows the verification of mainstream programming languages and focuses entirely on the communication patterns, while the effects of the associated processes are summarized directly in each thread's pre- and postcondition.

1.2 Contributions

Unlike 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 solutions on delegation [36], our proposal uses the same send/receive channel methods for sending values, data structures, and channels. For example, [36] 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 specialised 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. [87] 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. Their logical formulae on protocols can also be localised to each channel and may be freely passed through procedural boundaries, but similar to session types, their logic requires a special primitive for external choice and additionally does not support the verification of optimal protocols without labels.

Furthermore, 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 verification of communication safety. Furthermore we can 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, yet ensure that they remain compatible at each join point so as to prevent intra-channel deadlocks. More realistically, we also assume the presence of asynchronous communication protocols, where send commands are non-blocking.

In what follows we will summarize our contribution:

- **Session Logic:** In chapter 3 in accordance with our hypothesis we presented a novel session logic with disjunctions to specify and verify the implementation of the communication protocols. This work was presented in [29, 70] and is being implemented on top of the HIP/SLEEK system [26]

- **Soundness proof:** In chapter 4 we proved the soundness of our theory.
- **Automatic verification tool:** In chapter 5 we presented our *SESSION-HIP-SLEEK* toolchain, developed during the course of the PhD. The toolchain is implemented in *Objective Caml (OCaml)* and consists of two parts an entailment proofer, namely *SESSION-SLEEK* and the verification tool, namely *SESSION-HIP*. The tools are built on top of the *HIP-SLEEK* toolchain, and it facilitates the automatic reasoning concerning the correctness of programs that use pointers and which communicate with other programs via channels.
- **Comparison with other approaches:** In chapter 6, we presented the six most competitive tools for the verification of protocols using session types. We have compared each of these tools with our tool, by giving a set of actual examples and pointing out the most significant differences.
- **Application in the railway industry:** In chapter 7, we have presented two possibilities for using our Session Logic in the railway software development industry. Firstly, we presented a complete development and verification method for the development of interlocking software using the geographical interlocking approach. The method of encoding the interlocking requirements and projection to the entities were presented in [71, 69]. Additionally, we present the applicability of our theory in the software development of automatic train protection systems. For demonstration purposes, we encoded into Session Logic a set of requirement from the TBL1+ specification provided by Siemens, and also three specifications from the OpenETCS.

In this thesis, we argue strongly for the simplicity, expressiveness and applicability of our logic by demonstrating it through a number of examples.

1.3 Related Works

Our work features an enforcement of protocol specification, via verification in a general purpose imperative programming language which fits into the protocols verification theory. We have identified three main research directions in order to address the challenge: automated model extraction, automated code generation and automated code verification.

The first method begins with the protocol code and extracts an abstract model on which the protocol properties can be verified [77, 3]. In the case of an error, the approach provides an abstract model for developers to identify the problem, but in general this model is too complex hence difficult to understand. Therefore the usability of such a verification is restricted to small problems where the model can be easily understood.

The second approach suggests a development method which, starting from a verified abstract model of protocol, generates a correct but incomplete implementation of the protocol [63, 99]. On the other hand, it is common for even the best programmers to make simple mistakes, and an extension which seems safe can still contain bugs, and in such cases the correctness of the protocol implementation cannot be ensured.

Our work adheres to the automatic code verification direction. Due to the huge amount of work in this direction, we focus on the key differences between related works on static verification of high order protocols.

First of all we consider session types, which is a typing discipline for ensuring the communication safety of distributed programs, originally developed in the π-calculus [113, 24] and later extended to functional and object-oriented languages [60, 105]. The main idea of session types is that applications are built starting from units of design called session models. Existing implementations of session types [60, 105] are focused on static type checking of endpoint processes against these local session type specifications. The direct application of the theoretical session type techniques to the current practice, however, presents a few obstacles. First, the existing type systems are targeted at calculi and programming languages with first class primitives for linear communication channels and communication-oriented control flows.

For example, the work in [36] proposes a type discipline to prove the correctness of session types in an object oriented programming language, but their type system requires a programming language with a set of communication primitives, as:*send*, *receive*, *sendIf*, *receiveIf*. Trying to eliminate *sendIf*, *receiveIf* from their formal language makes their theory unsound. Other works such as [52, 62, 35, 100, 86] suffer from the same problem. The main problem with this theory is the limited applicability caused by the session types control flows, which has been identified in several works [63, 97]. Having a verification method which targets languages with such primitives is not so useful from an industrial point of view. The reason these languages have such primitives is presented in several works [31, 60]. These type systems address specific forms of programming language which were directly built with the scope of implementing distributed systems; our more general approach aims at ensuring the correct behaviour of mainstream programming languages with respect to a more expressive session logic specification, through static verification of these languages.

Because of the limitations of these type systems, there are several works [23, 25, 11, 88, 98, 59] which enforce the session types specification by dynamic verification. The work in [23] presents a monitor-based information-flow analysis in multi-party sessions. An informal approach to monitoring based on multi-party session types, and an outline of monitors are presented in [25]. These works address the dynamic verification of the protocol specifications but their verification is not exhaustive and can not be applied at an early stage of development.

The relation between the π-calculus and separation logic is studied in [111] and [57], but their work provides a treatment of the π-calculus based on the semantic theory of separation logic, without concentrating on protocol verification. The same idea was studied also in Hoare logic in [89].

From the perspective of other protocol specification languages, there is one work [74] which attempts to encode the CSP (communicating process algebra) into Hoare Logic, but they have encoded only the send and receive commands without the branching, and without handling aliases and method calls.

Additionally, [12] suggests a logic to extend multi-party session specifications, by enriching the assertion language studied in [13] with the capability of referring to virtual states local to each network principal. Lately, Villard et al. [87, 123] 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.

From the tool perspective, other session-based tools, such as MOOL [115], MOOSE [95], Bica [114], SessionJava [60] based on type-states and SessionC [100], ParTypes [116] based on indexed dependent types for parallel programs also require syntax extensions or annotations to be implemented as static typing for most mainstream languages.

In summary, compared to these related works, our contribution focuses on the enforcement of global safety, by verifying the source code correctness with respect to its protocol in a general purpose imperative language.

From the railway industry perspective, there are several works which attempt to apply process calculi as CSP [94, 93, 66, 92, 109, 126] for modelling and verifying some aspects of railway interlocking systems. For example, [66, 93] proposes a technique to generate a *CSP* || *B* model from an interlocking scheme plane with their *OnTrack* tool, which can be verified with the *ProB* model checker. [126] also proposes a modelling technique based on CSP to verify whether the functional specification of a track layout which is given in a *control table* respects all the *signalling principles* or not. Despite this interesting approach, their mode-checker does not scale-up sufficiently for actual systems. Additionally, none of the previous verification methods target the verification of the final source code, and do not scale up sufficiently for realistic systems. Therefore, the problem of verifying large-scale interlocking is considered to be a real problem which must be solved [124, 47].

With this in mind, the second goal of this thesis is to address the above issues by enabling the static verification of communicating behaviour of geographic interlocking entities via our session logic predicate.

The aim of our work is to capture the verification of railway protocols in mainstream imperative languages, providing better support for heterogeneous distributed systems, and to achieve this by allowing components to be independently verified statically, while retaining the strong global safety properties of a verified homogeneous system. Our framework is based on the idea that, if each endpoint is independently verified statically to conform to their local protocols, then the global protocol is respected as a whole.

1.4 Structure of the Thesis

1.5 Publications

Chapter 2
Background

2.1 π-calculus

2.2 Session Types

2.3 Separation Logic

2.3.1 Hoare Logic

Soundness proof for Hoare Logic

2.3.2 Sequential Separation Logic

2.3.3 Concurrent Separation Logic

2.4 Summary

In this chapter we introduce the basic theories required for the understanding of this thesis. Firstly, we present π-calculus, a modern concept for modelling concurrent processes mathematically. Then the notion of session types is presented, this is a type discipline for regulating the communication behaviour of processes. Finally, we present separation logic, which is a new theory for the verification of sequential and concurrent programs with mutable states and aliasing.

Chapter 3
Session Logic

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 the Buyer is satisfied with the price, she sends the address as an object of type `Addr` and the Seller sends back the delivery date as an object of type `Date`. Otherwise, the Buyer quits the conversation. This example is modelled as a 2-party session in Fig. 3.1. 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:

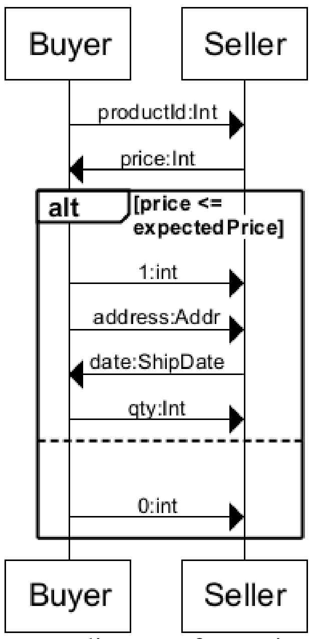

Fig. 3.1 Sequence diagram for an item purchasing

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

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

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

In the above, `!t` denotes the output of a value of type `t`, dually for `?t` which denotes input instead. The type `!{ok : ..., quit : ...}` denotes an internal choice (decision based on local values) of the options, while the type `?{ok : ..., 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 [60] 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);         |    int qty = receive(c);
  send(c,3);                        |   }
  }} 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:

```
buyer_ch   ≡ !String;?int;((!1;!Addr;?Date;!int)∨!0)
seller_ch  ≡ ~buyer_ch
           ≡ ?String;!int;((?1;?Addr;!Date;?int)∨?0)
```

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,

[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`.

instead of types or values, we allow more general properties (including ghost properties) to be passed into the 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.

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

Note that our channel specification uses several abbreviated notations. ?1 is a short-hand for $?\texttt{r}\cdot\texttt{r:int}\wedge\texttt{r}{=}1$, while !String is a shorthand for $!\texttt{r}\cdot\texttt{r:String}\wedge\texttt{true}$. 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 $?\texttt{r:int}\cdot\texttt{r}{>}1$ as a short-hand for $?\texttt{r}\cdot\texttt{emp}\wedge\texttt{r:int}\wedge\texttt{r}{>}1$. Another issue worth noting is that thread specification and channel specification need not be identical. As an example, let us provide a stronger specification for the seller's communication with the protocol, by insisting that price of products sold by this seller is at least 10 units, as follows:

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

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.

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`.

```
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);                int qty = receive(c);
  send(c,3);                              } else
 } else send(c,0);                         assert usr_opt = 0;
}                                        }
```

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 $\mathscr{C}(\texttt{c},\texttt{buyer\_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{buyer\_chan})$ which trivially succeeds. For the seller process, we would require $\mathscr{C}(\texttt{c},\texttt{seller\_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller\_sp})$. This second entailment also succeeds because the subsumption for sending operation is contravariant, as illustrated below.

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

Before a channel is used, it must first be opened by `open(c)` together with an appropriate channel specification. In contrast to previous work (such as [123] 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.

3.1 A Process Model for Sessions Logic

3.1.1 Asynchronous Session π-calculus with Assertion

3.2 SESSION-HIP

In order to prove our theory, we provide an imperative programming language: SESSION-HIP, which, while suited to the level of programmers, is simple enough to allow the proving of its properties.

3.2.1 SESSION-HIP Syntax

We formalize our approach on a concurrent imperative language enhanced with communication primitives. The syntax of the language is presented in Fig.3.2. Our language is an extension of the sequential language from [27]. 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. 3.5. 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 minimizing 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.

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 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
```

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 $\mathtt{S}_2$.

program definition	$Prog$	$::= tdecl^* \ meth^*$
type declaration	$tdecl$	$::= datat \mid spred$
data type	$datat$	$::= \texttt{data} \ c \ \{ \ (t \ v)^* \ \}$
types	t	$::= c \mid prim \mid Chan \mid dyn$
primitive types	$prim$	$::= \texttt{int} \mid \texttt{bool} \mid \texttt{void}$
method definition	$meth$	$::= t \ mn \ (ref \ (t \ v)^*, (t \ x)^*) \ mspec \ \{e\}$
enpoint	che	$N \mid D$
expressions	e	$::= null \mid k^{prim} \mid v \mid v.f \mid v := e \mid v_1.f := v_2 \mid e_1;e_2$ $\mid \texttt{if} \ (v)\texttt{then} \ e_1 \ \texttt{else} \ e_2 \mid \texttt{t} \ v; \ e \mid mn(v^*;x^*)$ $\mid \texttt{new} \ c(v^*) \mid \texttt{free}(v)$ $\mid (v_l^*, (vc_l = red \ c_l \ che)^*)\{e_1\} \mid\mid (v_r^*, (vc_r = red \ c_r \ che)^*)\{e_2\}$ $\mid \texttt{open}(c_1, c_2) \ \texttt{with} \ spred \mid \texttt{close}(c_1, c_2)$ $\mid \texttt{send}(c, v) \mid \texttt{receive}(c)$

Fig. 3.2 A Concurrent imperative language with sessions.

```
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 language 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:

$$\mathtt{S_3} \equiv \mathtt{!Object;(S_3 \vee !0)}$$

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:

$$\mathtt{S_4} \equiv \mathtt{!Object;(!1;S_4 \vee !0)}$$

Let now define the operational semantics of this language.

3.2.2 Operational Semantic

In this section we present an execution environment based on a small-step operational semantic for our language given as follows:

$$State \triangleq Stack \; x \; Heap \; x \; CHeap \qquad Stack \triangleq Var \rightarrow Val \cup Cell$$

$$CHeap \triangleq Endpoint \xrightarrow{\text{fin}} MQueue \; x \; MQueue \qquad Heap \triangleq Cell \xrightarrow{\text{fin}} Val$$

$$MQueue \triangleq QueueId \xrightarrow{\text{fin}} Val^*$$

The machine current state is represented by a tuple $\langle e,s,h,c\rangle$ where e denotes the command, s denotes the stack, h denotes the heap, and c denotes the current endpoint heap. The endpoint heap is unconventional compared to usual models of separation logic, therefore we will examine it in greater depth.

In our system, each channel has two endpoints which are owned in general by two different threads. The communication between two endpoints is modelled with a pair of queues which plays different roles for the two endpoints. The first endpoint considers the first queue as an input queue, while the second endpoint considers as an output queue, and vice versa for the second queue. This model offers an intuitive environment for the analyzes of communication between processes. As a consequence, the role of this heap is to store references to these queues. As a matter of fact, we want to remark that the separation of the standard heap from this heap is not necessary, but helps in the presentation of the semantic and simplifies the demonstration of the soundness theorem.

In the next we will present informally the operational semantics of SESSION-HIP for the standard commands.

Having the configuration $\langle e,s,h,c\rangle$ as a representative of the actual state of a machine, then the reduction steps can be formalized as a transition of the form: $\frac{valid(s_1,h_1,c_1)}{\langle e_1,s_1,h_1,c_1\rangle\hookrightarrow\langle e_2,s_2,h_2,c_2\rangle}$. The reduction can be interpreted like this: if a machine has a state $\langle e_1,s_1,h_1,c_1\rangle$, the state of $\langle s_1,h_1,c_1\rangle$ fulfill the **valid** specification and we execute the expression e_1 then the machine will change its state to $\langle e_2,s_2,h_2,c_2\rangle$. On top of that reduction semantic, we present some notations, which are important to understand the semantic.

Notations 3.2.1. We introduced **skip** to denote the empty expression and $s[v\mapsto \upsilon]$ to denote a variable v which maps to a value υ. Additionally, $\perp$ to denote an undefined or unknown value, **k** to denote a constant and $\mathbf{ret}(\mathbf{v}^*,\mathbf{e})$ to model the outcome of call invocation, where **e** represents the residual code of the call. The operation $s\cdot[v\mapsto \upsilon]$ adds the variable v to stack with the value υ. The operation $s_1 = s_2 - s_3$ is a shorthand for $s_2 = s_1\cdot s_3$ and removes stack s_3 from the stack s_2.

Additionally, next, we will give a list of formal definitions for the non-standard operators. They plays an important role in proving the correctness of our verification rules.

Definition 3.2.1. (Disjoint union) The disjoint union of two partial functions f and g with the same co-domain D is:

$$dom(f) \cdot dom(g) \overset{\text{fin}}{\rightharpoonup} D$$

$$(f \cdot g)(x) ::= \begin{cases} f(x) & \text{if } x \in dom(f) \\ g(x) & \text{if } x \in dom(g) \end{cases}$$

Definition 3.2.2. (Disjoint sub-states) Two states $\langle s_1, h_1, c_1 \rangle$, $\langle s_2, h_2, c_2 \rangle$ are disjoint sub-states of $\langle s, h, c \rangle$ denoted $\langle s_1, h_1, c_1 \rangle \# \langle s_2, h_2, c_2 \rangle$, if the following conditions hold:

1. $dom(s_1) \cup dom(s_2) \subseteq dom(s) \wedge dom(s_1) \cap dom(s_2) = \emptyset$
2. $dom(h_1) = \bigcup_{v_i \in dom(s_1)} part(v_i, h) \quad dom(h_2) = \bigcup_{v_j \in dom(s_2)} part(v_j, h)$
   $dom(h_1) \cup dom(h_2) \subseteq dom(h) \wedge dom(h_1) \cap dom(h_2) = \emptyset$
3. $dom(c_1) = \bigcup_{v_i \in dom(s_1)} part(v_i, c) \quad dom(c_2) = \bigcup_{v_j \in dom(s_2)} part(v_j, c)$
   $dom(c_1) \cup dom(c_2) \subseteq dom(c) \wedge dom(c_1) \cap dom(c_2) = \emptyset$

Now, after we have defined the union of two mappings in definition 3.2.1 and the notion of disjoint and thread compatible sub-states in definitions 3.2.2 and **??**, we will provide our extension of the concurrent HIP operational semantic in Fig.3.3.

First, before providing other details about the language, let's go over these rules in an informal way:

- command **open(f)** creates a communication environment by allocating the necessary resources and connecting them properly. More precisely, it allocates two endpoints $\mathbf{l_1}, \mathbf{l_2}$ and two empty queues $\mathbf{q_1}, \mathbf{q_2}$, and associates $\mathbf{q_1}$ as input queue, $\mathbf{q_2}$ as output queue to $\mathbf{l_1}$, and vice versa for $\mathbf{l_2}$. The execution of this command is possible only if $\mathbf{l_1}, \mathbf{l_2}, \mathbf{q_1}, \mathbf{q_2}$ are not allocated.
- **close(f)** removes the endpoints and queues of channel **f** from the channel heap, if parameter **f** is a properly defined empty channel (it means that the endpoints must be duals and the queues must be empties).
- **send(f,v)** moves the resources referenced by **v** from the heap and channel heap into the output message queue, and removes **v** from the stack.
- **receive(f)** extracts the first variable from the queue and adds the resources corresponding to this variable into the standard heap and channel's heap.
- **red ch {L,D}** is a helper function which can be used to extract the endpoint from a channel variable. This function only helps the thread function to extract the endpoints, and cannot be used for other purpose.
- $(\mathbf{v_l^*}, (\mathbf{ch_l} = \textbf{red c L})^*)\{\mathbf{e_1}\} || (\mathbf{v_r^*}, (\mathbf{ch_r} = \textbf{red c D})^*)\{\mathbf{e_2}\}$ has one of the most complex operational semantic. The specification creates two scopes for the two threads, therefore the race free execution of

the threads is explicit and can be simply verified by analyzing the parameters of the two threads. If the parameters of the two threads are race free then the threads are race free. In additional, the channels passed to the two threads must be pairs, where each thread owns one endpoint of the channel. If all the previous specifications are fulfilled, the two threads are reduces independently.

3.3 Verification Principle

Our verification mechanism, based on abstract interpretation, is an extension of the Hoare-style forward verification, more precisely it is an extension of the separation logic from [27].

A schematic overview of our verification mechanism is shown in Fig.3.4. The system requires as input a set of functions with pre- and post-conditions, and additionally the predicates which are required by the previously mentioned pre- and post-conditions. The predicates can be of two types, session logic predicates and separation logic predicates. Provided that the above-mentioned requirements are met, we can verify that the source code of each function meets its specification or not. The verification is done systematically for each expression using the corresponding verification rules from subsection 3.3.2. An expression is considered to be correct if and only if its precondition can be met by the current symbolic state. In this case, the precondition is removed and the post condition is added to the current state. The proof obligations generated by software verification systems are discharged by the SESSION-SLEEK theorem prover.

This theorem prover plays an important role in the automatic verification but can be omitted for simplicity from the soundness proof, because its extension was minimal and its correctness is proven in the thesis. We will return to this theorem prover in the implementation chapter, but for now, we omit them. Next, we present our specification language.

3.3.1 Specification Language

We develop our session specification language on top of the specification language (in Fig. 3.5) from [27]. 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 $\mathscr{C}(\mathtt{v},\mathtt{S})$ where $\mathtt{S}$ can denote a sending communication, a receiving communication, a sequence of communication operations and a choice of communication operations. $\mathtt{S}$ 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

$$\frac{l_1, l_2, q_1, q_2 \notin dom(c)}{\begin{array}{l}\langle open(f_c), s[f_c \mapsto \perp], h, c\rangle \hookrightarrow \\ \langle skip, s[f_c \mapsto \upsilon(\mathcal{C}, ((\upsilon_1(\mathcal{C}, l_1), \upsilon_2(\mathcal{C}, l_2)))], h, \\ c \cdot [l_1 \mapsto (q_1, q_2), l_2 \mapsto (q_2, q_1), q_1 \mapsto \emptyset, q_2 \mapsto \emptyset]\rangle\end{array}}$$

$$\begin{array}{l}\langle close(f_c), s[f_c \mapsto \upsilon(\mathcal{C}, ((\upsilon(\mathcal{C}, l_1), \upsilon(\mathcal{C}, l_2)))], h, \\ c \cdot [l_1 \mapsto (q_1, q_2), l_2 \mapsto (q_2, q_1), q_1 \mapsto \emptyset, q_2 \mapsto \emptyset]\rangle \hookrightarrow \\ \langle skip, s[f \mapsto \perp], h, c\rangle\end{array}$$

$$\frac{\begin{array}{l}v, f \in dom(s) \quad s(v) = \upsilon_t(\mathcal{H}, \_) \vee s(v) = \upsilon_t(\mathcal{C}, \_) \\ (h_t, c_t) = part(v, h) \quad h_1 = h - h_t \quad c_1 = c - c_t \quad s_1 = s - [v] \\ s(f) = \upsilon(\mathcal{H}, l) \quad l \notin dom(c_t)\end{array}}{\begin{array}{l}\langle send(f, v), s, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2]\rangle \hookrightarrow \\ \langle skip, s_1, h_1, c_1 \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2 \oplus (\upsilon_t, h_t, c_t)]\rangle\end{array}}$$

$$\frac{v, f \in dom(s) \quad s(v) = \upsilon_t(\mathcal{K}, \_) \quad s_1 = s - [v] \quad s(f) = \upsilon(\mathcal{H}, l)}{\begin{array}{l}\langle send(f, v), s, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2]\rangle \hookrightarrow \\ \langle skip, s_1, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2 \oplus (\upsilon_t, \emptyset, \emptyset)]\rangle\end{array}}$$

$$\frac{s(f) = \upsilon(\mathcal{H}, l) \quad c_1 = c \cdot c_t \quad h_1 = h \cdot h_t \quad \upsilon_t(\mathcal{H}, \_) \vee \upsilon_t(\mathcal{C}, \_)}{\begin{array}{l}\langle receive(f), s, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto (\upsilon_t, h_t, c_t) \oplus m_1]\rangle \hookrightarrow \\ \langle \upsilon_t, s, h_1, c_1 \oplus [l \mapsto (q_1, q_2), q_1 \mapsto m_1]\rangle\end{array}}$$

$$\frac{s(f) = \upsilon(\mathcal{H}, l) \quad \upsilon_t(\mathcal{K}, \_)}{\begin{array}{l}\langle receive(f), s, h, c \cdot [l_1 \mapsto (q_1, q_2), q_1 \mapsto (\upsilon_t, h_t, c_t) \oplus m_1]\rangle \hookrightarrow \\ \langle \upsilon_t, s, h_1, c_1 \cdot [l_1 \mapsto (q_1, q_2), q_1 \mapsto m_1]\rangle\end{array}}$$

$$\frac{ch \in dom(s) \quad ch \mapsto \upsilon(\mathcal{C}, (\upsilon(\mathcal{C}, l_1), \upsilon(\mathcal{C}, l_2)) \quad l_1 \in dom(c) \quad l_1 = (q_1, q_2)}{\langle red\ ch\ N, s, h, c\rangle \hookrightarrow \langle \upsilon(\mathcal{C}, l_1), s, h, c\rangle}$$

$$\frac{ch \in dom(s) \quad ch \mapsto \upsilon(\mathcal{C}, (\upsilon(\mathcal{C}, l_1), \upsilon(\mathcal{C}, l_2)) \quad l_2 \in dom(c) \quad l_2 = (q_1, q_2)}{\langle red\ ch\ N, s, h, c\rangle \hookrightarrow \langle \upsilon(\mathcal{C}, l_2), s, h, c\rangle}$$

Fig. 3.3 Operational semantics of communication commands

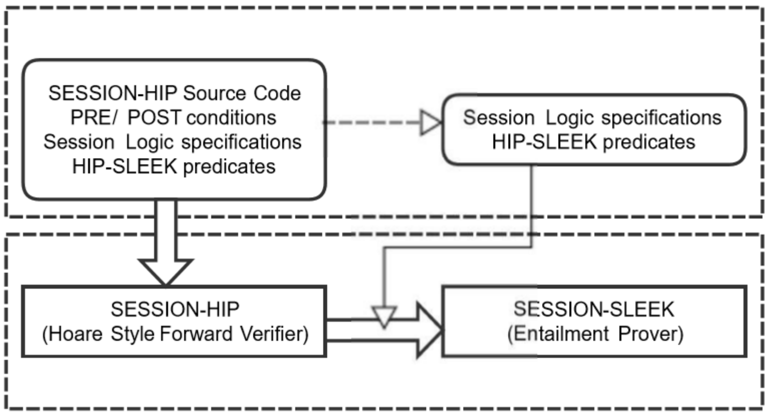

Fig. 3.4 SESSION-HIP-SLEEK verification principle

polyhedral 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 normalized to the Φ form [27].

Shape predicate	*spred*	::=	$p(\texttt{root},v^*) \equiv \Phi$
Formula	Φ	::=	$\bigvee \sigma^*$
	σ	::=	$\exists\, v^*\cdot\kappa\wedge\pi$
Method specification	*mspec*	::=	*requires* Φ_{pr} *ensures* Φ_{po}
Session formula	S	::=	$\texttt{emp} \mid ?r\cdot\Phi \mid !r\cdot\Phi \mid \sim S \mid S_1;S_2 \mid S_1 \vee S_2$
Heap formula	Δ	::=	$\Phi \mid \Delta_1\vee\Delta_2 \mid \Delta\wedge\pi \mid \Delta_1*\Delta_2 \mid \exists v\cdot\Delta$
	κ	::=	$\texttt{emp} \mid v\mapsto c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 \mid \mathscr{C}(v,S)$
	π	::=	$\gamma\wedge\phi$
	γ	::=	$v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1\wedge\gamma_2$
Pureformula	ϕ	::=	$r:t \mid \varphi \mid b \mid a \mid \phi_1\wedge\phi_2 \mid \phi_1\vee\phi_2 \mid \neg\phi \mid \exists v\cdot\phi \mid \forall v\cdot\phi$
	b	::=	$\texttt{true} \mid \texttt{false} \mid v \mid b_1=b_2$
	a	::=	$s_1{=}s_2 \mid s_1{\leq}s_2$
Presburger arithmetic	s	::=	$k^{\texttt{int}} \mid v \mid k^{\texttt{int}}\times s \mid s_1{+}s_2$ $\mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) \mid \lvert\texttt{B}\rvert$
	φ	::=	$v{\in}\texttt{B} \mid \texttt{B}_1{=}\texttt{B}_2 \mid \texttt{B}_1{\sqsubset}\texttt{B}_2 \mid \texttt{B}_1{\sqsubseteq}\texttt{B}_2 \mid \forall v{\in}\texttt{B}\cdot\phi \mid \exists v{\in}\texttt{B}\cdot\phi$
Bag constraint	B	::=	$\texttt{B}_1{\sqcup}\texttt{B}_2 \mid \texttt{B}_1{\sqcap}\texttt{B}_2 \mid \texttt{B}_1{-}\texttt{B}_2 \mid \emptyset \mid \{v\}$

Fig. 3.5 The specification language.

The semantic of this specification language is given in definition 3.3.1, in which the model relation $(s,h,c)|=\Phi$ denotes that the formula Φ evaluates to true in (s,h,c). In order to avoid confusion, we must mention that $s_1\#s_2$ denotes a stack where s_1 and s_2 are domain-disjoint. Additionally $s_1\cdot s_2$ indicates the union of two disjoint stacks s_1 and s_2. The operations # and $\cdot$ can be applied on the standard heap and channel heap with the same meaning.

Next, let us present this definition informally. The rule $s,h,c \models \Phi_1 \vee \Phi_2$ says that at least one from formulas Φ_1 and Φ_2 must be fulfilled by the state. $s,h,c \models \exists v_{1...n}\cdot\kappa\wedge\pi$ indicates the fact that the stack must fulfill π (the pure part of a formula) and the stack and heap must fulfill κ (the standard heap formula and channel heap formula). $s,h,c \models \kappa_1*\kappa_2$ points out that exists two disjointed heap parts h_1, c_1 and h_2, c_2, such that h_1, c_1 implies κ_1 and h_2, c_2 implies κ_2. $s,h,c \models \texttt{emp}$ supposes an empty heap. $s,h,c \models p \mapsto C(v_{1..n})$ states that C must be a definition of a data structure, and p must indicate to a heap location, wherein the fields of C are stored. Finally, $s,h,c \models \mathscr{C}(v,S)$ describes a channel heap location $s(v) = l$ which stores an input queue, an output queue and a session specification.

Definition 3.3.1. (Model of Separation Constraint)

$$
\begin{array}{ll}
s,h,c \models \Phi_1 \vee \Phi_2 & \text{if } s,h,c \models \Phi_1 \quad \vee \quad s,h,c \models \Phi_2 \\
s,h,c \models \exists v_{1\ldots n} \cdot \kappa \wedge \pi & \text{if } \exists v_{1\ldots n}, s = [v_1 \mapsto \mathrm{v}_1, \ldots, v_n \mapsto \mathrm{v}_n] \wedge s \models \pi \wedge s,h,c \models \kappa \\
s,h,c \models \kappa_1 * \kappa_2 & \text{if } \exists h_1, h_2, c_1, c_2,\ h_1 \# h_2 \wedge h_1 \cdot h_2 = h \wedge c_1 \# c_2 \wedge c_1 \cdot c_2 = c \\
 & s,h_1,c_1 \models \kappa_1 \wedge s,h_2,c_2 \models \kappa_2 \\
s,h,c \models \texttt{emp} & \text{if } dom(h) = \emptyset \wedge dom(c) = \emptyset \\
s,h,c \models p \mapsto C(v_{1..n}) & \text{if } \exists l, f_1, \ldots, f_n \quad s(p) = l \quad data\, C\{t_1\ f_1, \ldots, t_n\ f_n\} \in P \\
 & \wedge\ h[l \mapsto C[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)]] \\
s,h,c \models \mathscr{C}(v,S) & \text{if } \exists l, q_i, q_o \quad s(v) = l \wedge c[l \mapsto (q_i, q_o)]
\end{array}
$$

We are now ready to present our verification rules.

3.3.2 Verification Rules

In the next we will present, the verification rules of our concurrent separation logic. The fundamental notion of our verification mechanism is the Hoare triple. A triple $\{\Delta_1\}e\{\Delta_2\}$ from Fig.3.6 describes how the execution of an expression e changes a logical state which corresponds to Δ_1 into a logical state which corresponds to Δ_2. In order to define our verification rules, we need the following notations:

Notations 3.3.1. We use e to denote an expression. Additionally, we use $\vdash \{\Delta_1\}e\{\Delta_2\}$ to denote a standard Hoare triple, where Δ_1 is the precondition and Δ_2 is the post-condition.

After we have all the ingredients, we will provide an informal description of our formal verification rules to help the reader to understand it without difficulties.

Let us focus on communication rules from Fig. 3.6:

- **[OPEN]** According to this rule, the function *open* allocates two endpoint locations, and associates them with two session logic specifications. More precisely, the original protocol specification which decorates the open will be associated to the first endpoint while the dual of the previous specification will be associated to the second endpoint.
- **[CLOSE]** The rule verifies that the variable which is given as an argument to the *close* function indicates to a tuple which has two references which point to two different endpoints. The session specification of these endpoints must indicate an empty protocol, which is a necessary requirement according to our process algebra. If the previous conditions hold, than the channel is deallocated, and it is removed from the heap.
- **[SEND]** The precondition of this rule requires that the endpoint and the message corresponding to variables f and v to be present in the current state. Moreover, the transmission of this message must be the next expected step according to the protocol specification of this endpoint.

[SEND]

$$\frac{\begin{array}{l}\Phi_v = reach(v, \Delta_1) \\ \Delta_1 = \Delta * \mathscr{C}(f, \bigvee_{i \in I} !r \cdot \Phi_i; S_i) * \Phi_v \\ \exists j \in I \quad \Phi_v \vdash [v/r]\Phi_j \qquad \Delta_2 = \Delta * \mathscr{C}(f, S_j)\end{array}}{\vdash \{\Delta_1\} send(f, v) \{\Delta_2\}}$$

[RECEIVE]

$$\frac{\begin{array}{l}\Delta_1 = \Delta * \mathscr{C}(f, \bigvee_{i \in I} ?r \cdot \Phi_i; S_i) \\ \Delta_2 = \bigvee_{j \in I} \Delta * \mathscr{C}(f, S_j) * [res/r]\Phi_j\end{array}}{\vdash \{\Delta_1\} receive(f) \{\Delta_2\}}$$

[OPEN]

$$\frac{\Delta_1 = \Delta * \mathscr{C}(p_1, S) * \mathscr{C}(p_2, \sim S) \wedge f = (p_1, p_2)}{\vdash \{\Delta\} open(f)\ with\ S \{\Delta_1\}}$$

[CLOSE]

$$\frac{\Delta_1 = \Delta * \mathscr{C}(p_1, semp) * \mathscr{C}(p_2, semp) \wedge f = (p_1, p_2)}{\vdash \{\Delta_1\} close(f) \{\Delta\}}$$

Fig. 3.6 Session Logic primitives verification rules

This also means that the message must be compliant with one of the logical specification required by the protocol. As can be anticipated, the access to the transmitted data is lost after sending, and the protocol specification of the endpoint is changed according to the labelled transition semantics. As a consequence, a program which was considered to be correct after a verification cannot access the resources attached to a message after it has been sent; only the recipient of the message will be able to further access it, once it has received the message.

- **[RECEIVE]** In contrast with *send* the *receive* rule is more simple, and requires only a protocol specification which starts with a set of *receive* operations. If this precondition is fulfilled, for the endpoint pointed by f, then the receive operation is valid and the symbolic state can be changed. The change itself consists of consuming all the aforementioned receive specifications from the protocol and adding their logical specifications as a disjunction to the current state. By this adding, we have covered all possible receive actions.

From a sequential execution aspect, the *send* operation can be seen as a complex disposal, which can be executed only if the memory location fulfils some logical criteria, whereas the *receive* can be seen as a bulk allocation where the allocated memory holds some previously specified logical specification.

Chapter 4
Soundness Proof

4.1 Concurrent Operational Semantics

4.2 Contract Obedience Proof

4.3 Soundness Proof

4.4 Summary

In this chapter we extend the operational semantic from chapter 2, in order to provide an adequate framework for proving the correctness of our verification mechanism. Then, we will provide the necessary properties in the form of theorems to prove the correctness of our verification. Finally, we will provide a proof of soundness for our verification.

Chapter 5
SESSION-HIP-SLEEK

In this chapter we present our *SESSION-HIP-SLEEK* toolchain, developed during the course of the thesis. The toolchain is implemented in *Objective Caml (OCaml)* and consists of two parts an entailment proofer, namely *SESSION-SLEEK* and the verification tool, namely *SESSION-HIP*. The tools are built on top of the *HIP-SLEEK* toolchain, and it facilitates the automatic reasoning concerning the correctness of programs that use pointers and which communicate with other programs via channels. These tools accept as an input a file name and a set of options and it produces a textual output. More precisely the *SESSION-SLEEK* tool accepts as input a file with the extension *slk* and a set of options. The file can contain a set of protocol specifications, a set of separation logic predicate and a set of entailment checks requirements. For this file, the *SESSION-SLEEK* can produce a parser error or a set of result for each entailment. If the entailment is valid it produces an *OK* message, otherwise, it shows the entailment which can not be proved. The tool allows us to have special instructions in the file for displaying more detail about a proof. The *SESSION-HIP* tool requires as input a file with the extension *ss* and also a set of options. The file should contain a set of separation logic predicates, a set of protocol specifications and a set of function with pre- and post-conditions written using the syntax of *SESSION-HIP*. If the syntax is correct then the tool produces for each function an output. The output can be a *SUCCESS* message if the function is correct according to its pre and post-conditions or an error message if the function has an error. The error message gives the necessary information as the line of code and the entailment rule which cannot be proven, to help the developer to debug the program. Next, we will present this tools in more detail.

5.1 SESSION-SLEEK Entailment Prover

This section is dedicated to present our SESSION-SLEEK theorem prover. As can be seen in Fig.5.1, the role of this tool is to discharge obligations generated by SESSION-HIP software verification systems. SESSION-HIP generates formulas which are a combination of session logic, separation logic, and first order logic. As can be expected, none of the existing provers can handle such formulas. To prove this, SESSION-SLEEK uses also a set of on the shelf theorem provers in the background, but it has also a set of own proving mechanisms implemented in it. A formula generated by SESSION-HIP it looks like bellow:

$$\Delta' \vdash^{\kappa}_{V} \Delta * R \qquad (1)$$
$$\kappa * \Delta' \vdash \exists V \cdot (\kappa * \Delta) * R \qquad (2)$$

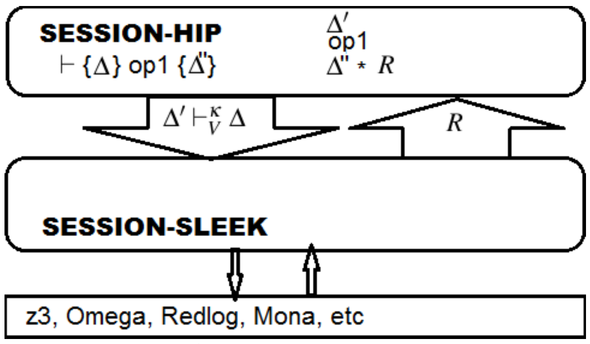
SESSION-HIP
⊢ {Δ} op1 {Δ''}
Δ'
op1
Δ'' * R
$\Delta' \vdash^{\kappa}_{V} \Delta$
R
SESSION-SLEEK
z3, Omega, Redlog, Mona, etc

Fig. 5.1 SESSION-SLEEK

According to the previous rule (1) [1], the proving process of the heap entailment consists of checking that the antecedent specification Δ' is enough accurate to meet the consequence Δ and to infer the frame context R. Note that, V stores a set of existentially quantified variables, and k stores heap locations from the antecedent, which are used to connect the heap locations in the consequence. We also want to mention, that since the *SLEEK* prover is not complete, exists cases when an entailment is possible but it can not be proved by this prover.

Since there are a lot of entailment rules in the *SLEEK*, and because the presentation of this rules can not contribute significantly to the understanding of this thesis, and being presented very well in several articles as [32, 27], we omit to present them here in detail.

5.1.1 Entailment of the Send and Receive

In the next we will present our entailment rules. In our work we extend the separation logic prover SLEEK [27] to prove whether one session logic formula Δ' in the combined abstract domain entails another one $\Delta{:}\Delta'{\vdash}\Delta{*}R$. R is called the *frame* which is useful to support sub-structural reasoning rules of separation logic. We extended the SLEEK rules to support entailment over the session logic formulae (see Fig. 5.2). 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 entailment rules coming from the compatibility of two session logic specifications. The rules are given in Fig. 5.3.

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.

[1]Note, that (1) is an alternative to more simple representation of (2)

$$\text{[OUTPUT]}\quad \frac{\Delta_2 \vdash \Delta_1}{!r \cdot \Delta_1 \vdash !r \cdot \Delta_2} \qquad \text{[INPUT]}\quad \frac{\Delta_1 \vdash \Delta_2}{?r \cdot \Delta_1 \vdash ?r \cdot \Delta_2}$$

$$\text{[SEQ-CHAN]}\quad \frac{e_1 \vdash e_2 \quad rest_1 \vdash rest_2}{e_1; rest_1 \vdash e_2; rest_2} \qquad \text{[MATCH-CHAN]}\quad \frac{S_1 \vdash S_2}{\mathscr{C}(c, S_1) \vdash \mathscr{C}(c, S_2)}$$

Fig. 5.2 Entailment rules for session logic.

$$\text{[OUTPUT-OR]}\quad \frac{\Delta_2 \vdash \Delta_1 \quad \neg(\Delta_3 \wedge \Delta_1)}{!r_1 \cdot \Delta_1 \vee !r_2 \cdot \Delta_3 \vdash !r_1 \cdot \Delta_2} \qquad \text{[INPUT-OR]}\quad \frac{\Delta_1 \vdash \Delta_2 \quad \neg(\Delta_3 \wedge \Delta_2)}{?r_1 \cdot \Delta_1 \vdash ?r_1 \cdot \Delta_2 \vee ?r_2 \cdot \Delta_3}$$

Fig. 5.3 Entailment rules for session logic disjunction.

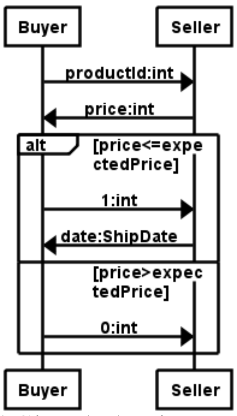

Fig. 5.4 Simple business protocol

We present out entailment rules 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 an accept message and Seller sends back the delivery date as an object of type `Date`. Otherwise, the Buyer quits the conversation. This example is modelled as 2-party session in Fig. 5.4.

We can summarize this Buyer-Seller protocol by using the following session logic specifications:

$$\begin{aligned}\texttt{buyer\_chan} &\equiv \texttt{!String}; \texttt{?r:int} \cdot \texttt{r>5}; ((\texttt{!1}; \texttt{?Date}) \vee \texttt{!0})\\ \texttt{seller\_chan} &\equiv \sim\texttt{buyer\_chan}\\ \texttt{buyer\_chan} &\equiv \texttt{?String}; \texttt{!r:int} \cdot \texttt{r>5}; ((\texttt{?1}; \texttt{!Date}) \vee \texttt{?0})\end{aligned}$$

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:

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

and let us consider the implementation from Fig.5.5.

```
                 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_sp)           |   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 = getBudget();      |  int usr_opt = receive(c);
 if (price <= budget) then{        |  if (usr_opt==1){
  send(c,1);                       |   ShipDate sd = getDate(p);
  ShipDate sd = receive(c);        |   send(c,sd);
 } else send(c,0);                 |  } else
}                                  |   assert usr_opt = 0;
                                   |  }
```

Fig. 5.5 Simple business protocol implementation.

In this application, the channel is opened in the main process by `open` which takes as argument the channel initial specification `buyer_chan`. 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 $\texttt{r}>10$ instead of $\texttt{r}>5$ 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 `seller` thread in our example, this means that $\mathscr{C}(\texttt{c},\texttt{seller\_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller\_sp})$. This relation can be proven by the SESSION-SLEEK using our $(OUTPUT)$ entailment rule as can be seen below:

$$\begin{array}{c}
\dots \\ \hline
((?1;!\texttt{Date})\vee?0) \vdash ((?1;!\texttt{Date})\vee?0) \\ \hline
\text{[OUTPUT]} \\
\texttt{r}>10 \vdash \texttt{r}>5 \\
!\texttt{r}\cdot\texttt{r}>5 \vdash !\texttt{r}\cdot\texttt{r}>10 \\
!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee?0) \vdash !\texttt{r:int}\cdot\texttt{r}>10;((?1;!\texttt{Date})\vee?0) \\ \hline
\text{[INPUT]} \\
?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee?0) \vdash ?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>10;((?1;!\texttt{Date})\vee?0) \\ \hline
\text{[FOLD]} \\
?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee?0) \vdash \texttt{seller\_sp} \\ \hline
\text{[UNFOLD]} \\
\texttt{seller\_chan} \vdash \texttt{seller\_sp} \\ \hline
\text{[MATCH-CHAN]} \\
\mathscr{C}(\texttt{c},\texttt{seller\_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller\_sp})
\end{array}$$

The entailment process should be read from bottom to top. The first rule $[MATCH-CHAN]$ from Fig.5.2 tries to successively match channels that can be proven to be same. In our case, by applying this rule we find that $\mathscr{C}(c, seller\_chan)$ and $\mathscr{C}(c, seller\_sp)$ are the same, therefore, the entailment process can be reduced to $seller\_chan \vdash seller\_sp$. Next, by applying the rule of $[UNFOLD]$ from [27], we replace the $seller\_chan$ predicate name by its definition. In the same way we replace $seller\_ch$ with its definition, and continues entailment checking. Additionally, by applying the rule of $[INPUT]$ from Fig.5.2, we can reduce the protocol specification by consuming the $?String$ term from the protocol. The $[OUTPUT]$ from Fig.5.2 entailment also succeeds because the subsumption for sending operation is contravariant. The rest of the entailment succeeds, because the left and right part of the entailment is identical.

On the other hand, if we consider the following `buyer` process specification:

$$\texttt{buyer\_sp} \ \equiv \ \texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}\texttt{10};((\texttt{!1};\texttt{?Date})\vee\texttt{!0})$$

then we can observe, that it imposes a weaker property over the receive price, using $\texttt{r}{>}\texttt{0}$ instead of $\texttt{r}{>}\texttt{5}$ that was captured in the channel specification `buyer_chan`. For the `buyer` thread in our example, this means that $\mathscr{C}(\texttt{c},\texttt{buyer\_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{buyer\_sp})$. This relation can also be proven by the SESSION-SLEEK using our $(INPUT)$ entailment rule as below:

$$\begin{array}{c}
\ldots \\
\hline
((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \ \vdash \ ((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \\
\hline
[\text{INPUT}] \\
\texttt{r}{>}\texttt{5} \ \vdash \ \texttt{r}{>}\texttt{0} \\
\texttt{?r}\cdot\texttt{r}{>}\texttt{5} \ \vdash \ \texttt{?r}\cdot\texttt{r}{>}\texttt{0} \\
\texttt{?r:int}\cdot\texttt{r}{>}\texttt{5};((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \ \vdash \ \texttt{?r:int}\cdot\texttt{r}{>}\texttt{0};((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \\
\hline
[\text{OUTPUT}] \\
\texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}\texttt{5};((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \ \vdash \ \texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}\texttt{0};((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \\
\hline
[\text{FOLD}] \\
\texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}\texttt{5};((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \ \vdash \ \texttt{seller\_sp} \\
\hline
[\text{UNFOLD}] \\
\texttt{buyer\_chan} \vdash \texttt{buyer\_sp} \\
\hline
[\text{MATCH-CHAN}] \\
\mathscr{C}(\texttt{c},\texttt{buyer\_chan}) \ \vdash \ \mathscr{C}(\texttt{c},\texttt{buyer\_sp})
\end{array}$$

The entailment process is very similar to the previous one, therefore, we explain only the $[OUTPUT]$ entailment, which enforces for the logical restriction to be covariant, so the entailment $?r\cdot r{>}5 \ \vdash \ ?r\cdot r{>}0$ can be simply proven.

5.1.2 Entailment of the Internal and External Choices

Outside of these four entailment rules, we have two rules in Fig.5.3 which are coming from the compatibility rules. To illustrate these rules we provide an implementation in Fig.5.6, wherein the *buyer_spc* and the *seller_spc* can be defined as below:

$$\begin{array}{lcl}
\texttt{buyer\_spc} & \equiv & \texttt{!String;?r:int}\cdot\texttt{r>5;!1;?Date} \\
\texttt{seller\_spc} & \equiv & \texttt{?String;!r:int}\cdot\texttt{r>5;((?1;!Date)} \lor \texttt{(?2;!int)}\lor\texttt{?0)}
\end{array}$$

These two protocols are compatible, according to the compatibility rules, therefore, our $SESSION-SLEEK$ tool provides the necessary entailment rules (Fig.5.3) to prove that the specifications of channels subsume the specifications of the threads.

The `buyer` process specification `buyer_spc` imposes a stronger protocol, using `!1;?Date` instead of $(\texttt{!1;?Date})\lor\texttt{!0}$. This means that the entailment $\mathscr{C}(\texttt{c},\texttt{buyer\_chan})\vdash\mathscr{C}(\texttt{c},\texttt{buyer\_spc})$ must be proven, by the SESSION-SLEEK using our $(OUTPUT-OR)$ entailment rule as can be seen below:

$$\begin{array}{c}
\ldots \\ \hline
\texttt{!1;?Date} \vdash \texttt{!1;?Date} \\ \hline
\text{[OUTPUT-OR]} \\
((\texttt{!1;?Date})\lor\texttt{!0}) \vdash \texttt{!1;?Date} \\ \hline
\text{[INPUT]} \\
\texttt{!r:int}\cdot\texttt{r>5;((!1;?Date)}\lor\texttt{!0)} \vdash \texttt{?r:int}\cdot\texttt{r>0!1;?Date} \\ \hline
\text{[OUTPUT]} \\
\texttt{!String;?r:int}\cdot\texttt{r>5;((!1;?Date)}\lor\texttt{!0)} \vdash \texttt{!String;?r:int}\cdot\texttt{r>0!1;?Date} \\ \hline
\text{[FOLD]} \\
\texttt{!String;?r:int}\cdot\texttt{r>5;((!1;?Date)}\lor\texttt{!0)} \vdash \texttt{seller\_spc} \\ \hline
\text{[UNFOLD]} \\
\texttt{buyer\_chan} \vdash \texttt{buyer\_spc} \\ \hline
\text{[MATCH-CHAN]} \\
\mathscr{C}(\texttt{c},\texttt{buyer\_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{buyer\_spc})
\end{array}$$

The entailment process is very similar to the previous ones, therefore, we explain only the $[OUTPUT-OR]$ entailment, which permits for the internal choice to have less branches as was originally specified, so the entailment $((!1;?Date)\lor!0)\ \vdash\ !1;?Date$ can be easily proven to be correct.

Finally, if we consider the `seller` protocol specification from Fig.5.6 then we can observe, that it imposes for the external choice to have more branches as was originally specified, so the entailment can be easily proven, as can be seen below:

```
void buyer(Chan c, String p)
  requires 𝒞(c, buyer_spc)
  ensures 𝒞(c, emp)
{ send(c, p);
 Double price = receive(c);
 send(c, 1);
 ShipDate sd = receive(c);
}
```

```
void seller(Chan c)
  requires 𝒞(c, seller_spc)
  ensures 𝒞(c, emp)
{ String p = receive(c);
 send(c, getPrice(p));
 int usr_opt = receive(c);
 if (usr_opt==1){
  ShipDate sd = getDate(p);
  send(c, sd);
 } else if (usr_opt==2){
  send(c, 5);
 } else
  assert usr_opt = 0;
}
```

Fig. 5.6 Compatible session logic processes.

$$\begin{array}{c}
\ldots \\ \hline
?1;!\mathtt{Date} \vdash ?1;!\mathtt{Date} \\ \hline
\text{[INPUT-OR]} \\
((?1;!\mathtt{Date}) \vee ?0) \vdash ?1;!\mathtt{Date} \\ \hline
\text{[OUTPUT]} \\
!\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}>5;((?1;!\mathtt{Date}) \vee ?0) \vdash !\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}>5;?1;!\mathtt{Date} \\ \hline
\text{[INPUT]} \\
?\mathtt{String};!\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}>5;((?1;!\mathtt{Date}) \vee ?0) \vdash ?\mathtt{String};!\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}>5;?1;!\mathtt{Date} \\ \hline
\text{[FOLD]} \\
?\mathtt{String};!\mathtt{r}{:}\mathtt{int}\cdot\mathtt{r}>5;((?1;!\mathtt{Date}) \vee ?0) \vdash \mathtt{seller\_spc} \\ \hline
\text{[UNFOLD]} \\
\mathtt{seller\_chan} \vdash \mathtt{seller\_spc} \\ \hline
\text{[MATCH-CHAN]} \\
\mathscr{C}(\mathtt{c},\mathtt{seller\_chan}) \vdash \mathscr{C}(\mathtt{c},\mathtt{seller\_spc})
\end{array}$$

5.2 SESSION-HIP

5.2.1 Full Expressivity of Separation Logic

5.2.2 Higher-Order Session Logic

Chapter 6
Comparison of Session Logic with Other Similar Approaches

6.1 SESSION-HIP-SLEEK vs Heap-Hop

6.2 SESSION-HIP-SLEEK vs Session C

6.3 SESSION-HIP-SLEEK vs ParTypes

6.4 SESSION-HIP-SLEEK vs Session Types Type-state Tools

6.4.1 SESSION-HIP-SLEEK vs MOOSE

6.4.2 SESSION-HIP-SLEEK vs Session Java

6.4.3 SESSION-HIP-SLEEK vs BICA

6.5 Summary

In this chapter, we presented the most competitive six tools for the verification of protocols using session types. We have compared each of these tools with our tool, by giving a set of actual examples and pointing out the most significant differences. In order to summarise the comparisons, we will present the most important differences into the Tables 6.1, 6.2.

The table 6.1 can be interpreted as follows:

- The first column contains the name of the tools.
- The second column of the table contains a tick (✓) if the tool has support for delegation or cross (✗) if has no support
- The third column contains a tick (✓) if the tool has support for internal and external choice or a cross (✗) if it has no support
- The fourth column contains a tick (✓) if the tool has support for loop or a cross (✗) if it has no support
- The penultimate column of the table contains a tick (✓) if the tool requires some special primitives for the internal or external choices or cross (✗) if such primitives

Table 6.1 Tool support 1

Tool	Delegation	Internal External Choice	Loop	Require Special Primitives	Copyless Message Passing
SESSION-HIP-SLEEK	✓	✓	✓	✗	✓
Heap-Hop	✗	✓	✓	✓	✓
Session C	✗	✓	✓	✓	✗
ParType	✗	✗	✓	✓	✗
MOOSE	✓	✓	✓	✓	✗
Session Java	✓	✓	✓	✓	✗
Bica	✓	✓	✗	✓	✗

Table 6.2 Tool support 2

Tool	Type Constraint	Logic Constraint on Data	Data Shape Predicate	Support Broadcast	Support Aliasing
SESSION-HIP-SLEEK	✓	✓	✓	✗	✓
Heap-Hop	✗	✓	✗	✗	✓
Session C	✓	✗	✗	✓	✗
ParType	✓	✗	✗	✓	✗
MOOSE	✓	✗	✗	✗	✗
Session Java	✓	✗	✗	✗	✗
Bica	✓	✗	✗	✗	✗

are not needed. (Note that if the verification tool requires these primitives then the language must have them, otherwise the verification does not work. So it is better if the tool does not require primitives.)

- The last column of the table contains a tick (✓) if the tool has support for the verification of copyless message passing or cross (✗) if has no support

The table 6.2 can be interpreted as follows:

- The first column contains the name of the tools.
- The second column of the table contains a tick (✓) if the tool has support for type checking or cross (✗) if has no support
- The third column contains a tick (✓) if the tool has support for logical constraint on the transmitted data or a cross (✗) if it has no support

- The fourth column contains a tick (✓) if the tool has support for using separation logic shape predicates to constraint the transmitted data or cross (✗) if it has no support
- The penultimate column of the table contains a tick (✓) if the tool has support for broadcasting or cross (✗) if it has no support
- The last column of the table contains a tick (✓) if the tool has support for the verification of programs with aliases or cross (✗) if has no support

Unlike previous approaches, we developed a tool based on session logic with a natural use of disjunction to specify and verify the implementation of communication protocols. Even though the logic used by the tool is based on two-party channel sessions, it can also handle delegation through the use of higher-order channels. Different from the tools from subsections 6.4.1, 6.4.2, 6.4.3, our proposal uses the same send/receive channel methods for sending values, data structures, and 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, all of the previously presented solutions require the host languages to be extended with a set of specialised switch constructs to model both internal and external choices. As a consequence, all of the previous approaches were restricted to languages with such control flows, which has reduced their applicability drastically. Additionally, our specification language is based on an extension of separation logic, and thus it supports heap-manipulating programs and copyless message passing. Comparing with the tool from subsection 6.1 we can observe, that their tool relies on state-based global contracts while our more general tool relies on a logic of session and it is built as an extension of separation logic with disjunction to support the standard control flows. In our case the logical formulae on protocols can also be localised to each channel and may be freely passed through procedural boundaries. Moreover, 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 specification 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.

Chapter 7
Application in the Railway Industry

7.1 Interlocking Modelling and Verification with Session Logic

7.1.1 Introduction

7.1.2 An illustrative Example

7.1.3 Encoding of the Requirements in Session Logic

7.1.4 Protocol Verification

7.1.5 Experimental Results

7.1.6 Conclusion

7.2 Automatic Train Protection Software Verification

7.2.1 Experimental Results

7.3 Summary

In this chapter, we have presented two possibilities for using our Session Logic in the railway software development industry. In the first case, we presented a complete development and verification method for the development of interlocking software using the geographical interlocking approach. The method of encoding the interlocking requirements and the projection to the entities were presented in [71, 69]. The verification of the source code and the experimental results are new. The result showed that the efficiency of our verification method is adequate for use in the railway industry. In the second use-case, we present the applicability of our theory in the software development of automatic train protection systems. For demonstration purposes, we encoded into Session Logic a set of requirement from the TBL1+ specification provided by Siemens, and also three specifications from the OpenETCS. We have implemented the corresponding source codes and the results have shown that this verification method is adequate for the verification of such source codes.

Chapter 8
Conclusion and Future Directions

With the explosive growth of wide area network infrastructures (such as the internet, GSM, etc.) in the last two decades, the development of distributed programs has increased dramatically, and these are used nowadays in nearly all domains: financial services, commercial services, entertainment, health care, telecommunication, defence, industrial automation etc.

On the other hand, the design and develop of these programs is quit difficult. The difficulties arise both in ensuring the safety correctness, as well as in obtaining high performance. From the safety perspective, we have to take into account a set of new issues, for example the sharing of common resources, or the synchronisation of these processes.

Despite these safety issues, these distributed programs are widely used in safety critical systems in industries such as flight control, air traffic control, industrial automation, automotive and railway industry, because of the separation concept which can increase reliability and scalability of these systems. As a result a modern car features more than 70 electronics control units [22] connected at least via 5 different bus systems, or a modern aircraft uses at least seven computers only for the fly-by-wire systems [19].

In this thesis, we addressed this problem of verifying distributed programs in safety critical environments, by encoding the communication behaviour of these systems into our protocol specification language, (session logic), and verifying automatically the source code correctness in accordance with these specifications.In what follows, we will detail the main contributions of this thesis. The contributions are highlighted in the following:

- **Session Logic:** In chapter 3 in accordance with our hypothesis we presented 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 of naturally handling 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 was presented in [29, 70] and it is being implemented on top of the HIP/SLEEK system [26]

- **Soundness proof:** After defining our session logic, in chapter 4 we proved the soundness of our theory. For this purpose we define more precise operational semantics for our actual programming language. This semantic is strongly related to the initial one but it lets us explore the link between the programming language and its protocol specification. By offering this proof, we guarantee for the users of our tool a program which was verified by our tool in accordance with our theory, it respects for sure the session logic protocols associated to this program. This is an important step, allowing us to guarantee the reliability of our tool and theory.

- **Automatic verification tool:** In chapter 5 we presented our *SESSION-HIP-SLEEK* toolchain, developed during the course of the PhD. The toolchain is implemented in *Objective Caml (OCaml)* and consists of two parts an entailment proofer, namely *SESSION-SLEEK* and the verification tool, namely *SESSION-HIP*. The tools are built on top of the *HIP-SLEEK* toolchain, and it facilitates the automatic reasoning concerning the correctness of programs that use pointers and which communicate with other programs via channels. These tools accept as an input a file name and a set of options and it produces a textual output. More precisely the *SESSION-SLEEK* tool accepts as input a file with the extension *slk* and a set of options. The file can contain a set of protocol specifications, a set of separation logic predicate and a set of entailment checks requirements. For this file, the *SESSION-SLEEK* can produce a parser error or a set of result for each entailment. If the entailment is valid it produces an *OK* message, otherwise, it shows the entailment which can not be proved. The tool allows us to have special instructions in the file for displaying more detail about a proof. The *SESSION-HIP* tool requires as input a file with the extension *ss* and also a set of options. The file should contain a set of separation logic predicates, a set of protocol specifications and a set of function with pre- and post-conditions written using the syntax of *SESSION-HIP*. If the syntax is correct then the tool produces for each function an output. The output can be a *SUCCESS* message if the function is correct according to its pre and post-conditions or an error message if the function has an error. The error message gives the necessary information as the line of code and the entailment rule which cannot be proven, to help the developer to debug the program.

- **Application in the railway industry:** In chapter 7, we have presented two possibilities for using our Session Logic in the railway software development industry. In the first case, we presented a complete development and verification method for the development of interlocking software using the geographical interlocking approach. The method of encoding the interlocking requirements and the projection to the entities were presented in [71, 69]. The verification of the source code and the experimental results are new. The result showed that the efficiency of our verification method is adequate for use in the railway industry. In the second use-case, we present the applicability of our theory in the software development of automatic train protection systems. For demonstration purposes, we encoded into Session Logic a set of requirement from the TBL1+ specification provided by Siemens, and also three specifications from the OpenETCS. We have implemented the corresponding source codes and the results have shown that this verification method is adequate for the verification of such source codes. Of course, the applicability of our theory is not limited to these use-cases, and can be applied whenever the correctness of the communication must be checked. A potential application could be also the verification of the CBTC protocols implementation.

- **Comparison with other approaches:** In chapter 6, we presented the most competitive six tools for the verification of protocols using session types. We have compared each of these tools with our tool, by giving a set of actual examples and pointing out the most significant differences.

  Unlike previous approaches, we developed a tool based on session logic with a natural use of disjunction to specify and verify the implementation of communication protocols. Even though the logic used by the tool is based on two-party channel sessions, it can also handle delegation through the use of higher-order channels. Different from the tools from subsections 6.4.1, 6.4.2, 6.4.3, our proposal uses the same send/receive channel methods for sending values, data structures, and 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, all of the previously presented solutions require the host languages to be extended with a set of specialised switch constructs to model both internal and external choices. As a consequence, all of the previous approaches were restricted to languages with such control flows, which has reduced their applicability drastically. Additionally, our specification language is based on an extension of separation logic, and thus it supports heap-manipulating programs and copyless message passing. Comparing with the tool from subsection 6.1 we can observe, that their tool relies on state-based global contracts while our more general tool relies on a logic of session and it is built as an extension of separation logic with disjunction to support the standard control flows. In our case the logical formulae on protocols can also be localised to each channel and may be freely passed through procedural boundaries. Moreover, 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 specification 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. This work was partially presented in [68].

8.1 Future Directions

Having these results, we can think on several directions for further research in verifying parallel programs which uses message passing for synchronization and data exchange. A very interesting research direction would be to extend our session logic to multi-party and multi-channel specification. As can be observed, a simple encoding of multi-party communication into our session logic specification language is trivial, and it simply can be done by using a data structure wherein one field plays the role of a channel. The problem is that the API-s of the operation systems are a little bit different. For example the connection to the different communication parties must be done sequentially, or we can not wait for messages on different channels by using the standard "read" in a single thread. For this reason the API-s provides a set of special functions. For example in a Linux we have: "select", "pselect" [85], "poll", "ppoll" [84] and "epoll" [83] . Therefore, in order to have support for these API-s, our session logic must be extended in this sense. An additional interesting topic in this area would be to investigate the possible combining of our session logic with [12]. Their logic tries to summarize the effects of processes involved in the protocol, and to enforce it in a refined version of the multiparty session π-calculus. We think that it would be interesting to investigate the verification of these specification in a more realistic programming language as for example "SESSION-HIP".

Another interesting research topic would be to investigate the applicability of our theory in the object oriented programming paradigm. For example, we can investigate the encoding and verification of the calling order of methods in classes, similar as in [52]. The benefit of this verification would be the applicability of this theory for mainstream object oriented programming languages.

As a technical research topic we suggest the development of an IDE (integrated development environment) for the railway industry. These should provide a user friendly specification method for the encoding of session logic specification (For example in form of sequence diagrams with annotations), and also support for the development and verification of geographic interlocking software. This should be based on our theory and tools. For the interlocking software development we suggest a domain specific language as EURIS or LARIS [50]. The verification should be done automatically using our verification tool from chapter 5, and following our verification mechanism presented in section 7.1.

Another interesting topic would be the verification of a large set of linux drivers which control some peripheral units using a set of messages transmitted via the front side bus (FSB) or nowadays via the Platform Controller Hub in case of an Intel architecture or via SPI or I2C or other peripheral buses in other RISC architectures([78–82]) .

The combination of two state of the art theory, namely separation logic and session types into our coherent session logic framework has opened too many possibilities to be enumerated here, so we let the reader to discover and explore the rest of the possibilities by itself. In order to help the reader to discover these opportunists, we would like to close this section with a suggestion: In informatics, the communication can be found approximately everywhere just we need to look at the problem carefully.

References

[1] Abdulla, P. A., Deneux, J., Stålmarck, G., Ågren, H., and Åkerlund, O. (2004). Designing safe, reliable systems using scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer. (page(s): [1], [2])

[2] Abrial, J.-R. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press. (page(s): [1])

[3] Bai, G., Lei, J., Meng, G., Venkatraman, S. S., Saxena, P., Sun, J., Liu, Y., and Dong, J. S. (2013). Authscan: Automatic extraction of web authentication protocols from implementations. In *NDSS*. (page(s): [5])

[4] Baltazar, P., Mostrous, D., and Vasconcelos, V. T. (2012). Linearly refined session types. *arXiv preprint arXiv:1211.4099*. (page(s):)

[5] Banci, M. and Fantechi, A. (2005). Geographical versus functional modelling by statecharts of interlocking systems. *Electron. Notes Theor. Comput. Sci.*, 133:3–19. (page(s): [2])

[6] Behm, P., Desforges, P., and Meynadier, J.-M. (1998). Météor: An industrial success in formal development. In *B'98: Recent Advances in the Development and Use of the B Method*, pages 26–26. Springer. (page(s): [1])

[7] Ben-Ari, M. (2001). The bug that destroyed a rocket. *ACM SIGCSE Bulletin*, 33(2):58. (page(s): [1])

[8] Bernardi, G., Dardha, O., Gay, S. J., and Kouzapas, D. (2014). On duality relations for session types. In *Trustworthy Global Computing*, pages 51–66. Springer. (page(s):)

[9] Bernardi, G. and Hennessy, M. (2014). Using higher-order contracts to model session types. In *CONCUR 2014–Concurrency Theory*, pages 387–401. Springer. (page(s):)

[10] Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., and Yoshida, N. (2008). Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008-Concurrency Theory*, pages 418–433. Springer. (page(s):)

[11] Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., and Yoshida, N. (2013a). Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 50–65. Springer. (page(s): [6])

[12] Bocchi, L., Demangeon, R., and Yoshida, N. (2013b). A multiparty multi-session logic. In *Trustworthy Global Computing*, pages 97–111. Springer. (page(s): [3], [7], [40])

[13] Bocchi, L., Honda, K., Tuosto, E., and Yoshida, N. (2010). A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010-Concurrency Theory*, pages 162–176. Springer. (page(s): [3], [7])

[14] Bonacchi, A. and Fantechi, A. (2014). On the validation of an interlocking system by model-checking. In Lang, F. and Flammini, F., editors, *Formal Methods for Industrial Critical Systems*, volume 8718 of *Lecture Notes in Computer Science*, pages 94–108. Springer International Publishing. (page(s): [2])

[15] Bonacchi, A., Fantechi, A., Bacherini, S., Tempestini, M., and Cipriani, L. (2013). Validation of railway interlocking systems by formal verification, a case study. In *Software Engineering and Formal Methods*, pages 237–252. Springer. (page(s):)

[16] Bono, V., Messa, C., and Padovani, L. (2011). Typing copyless message passing. In *Programming Languages and Systems*, pages 57–76. Springer. (page(s):)

[17] Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. (2005). Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM. (page(s):)

[18] Boulanger, J.-L., Fornari, F.-X., Camus, J.-L., and Dion, B. (2015). Scade: Language and applications. (page(s): [1])

[19] Brière, D. and Traverse, P. (1993). Airbus a320/a330/a340 electrical flight controls-a family of fault-tolerant systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 616–623. IEEE. (page(s): [37])

[20] Brookes, S. (2007). A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270. (page(s):)

[21] Brookes, S. D., Hoare, C. A., and Roscoe, A. W. (1984). A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599. (page(s): [3])

[22] Broy, M. (2006). Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM. (page(s): [37])

[23] Capecchi, S., Castellani, I., and Dezani-Ciancaglini, M. (2011). Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, pages 1–43. (page(s): [6])

[24] Castagna, G., Dezani-Ciancaglini, M., Giachino, E., and Padovani, L. (2009). Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 219–230, New York, NY, USA. ACM. (page(s): [6])

[25] Chen, T.-C., Bocchi, L., Deniélou, P.-M., Honda, K., and Yoshida, N. (2012). Asynchronous distributed monitoring for multiparty session enforcement. In *Trustworthy Global Computing*, pages 25–45. Springer. (page(s): [6], [7])

[26] Chin, W.-N., David, C., and Gherghina, C. (2011a). A hip and sleek verification system. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 9–10. ACM. (page(s): [1], [2], [4], [37])

[27] Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2012). Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036. (page(s): [14], [20], [21], [27], [30])

[28] Chin, W.-N., Gherghina, C., Voicu, R., Le, Q. L., Craciun, F., and Qin, S. (2011b). A specialization calculus for pruning disjunctive predicates to support verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 293–309. (page(s):)

[29] Craciun, F., Kiss, T., and Costea, A. (2015). Towards a session logic for communication protocols. In *ICECCS*. IEEE. (page(s): [4], [37])

[30] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer. (page(s): [1])

[31] Dardha, O., Giachino, E., and Sangiorgi, D. (2012). Session types revisited. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 139–150. ACM. (page(s): [6])

[32] David, C. and Chin, W.-N. (2011). Immutable specifications for more concise and precise verification. *SIGPLAN Not.*, 46(10):359–374. (page(s): [27])

[33] DeLine, R. and Fähndrich, M. (2001). Enforcing high-level protocols in low-level software. *ACM SIGPLAN Notices*, 36(5):59–69. (page(s): [3])

[34] Dezani-Ciancaglini, M. and De'Liguoro, U. (2009). Sessions and session types: An overview. In *Web Services and Formal Methods*, pages 1–28. Springer. (page(s):)

[35] Dezani-Ciancaglini, M., Giachino, E., Drossopoulou, S., and Yoshida, N. (2007). Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 207–245. Springer Berlin Heidelberg. (page(s): [6])

[36] Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., and Drossopoulou, S. (2006). Session types for object-oriented languages. In Thomas, D., editor, *ECOOP 2006 â€“ Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer Berlin Heidelberg. (page(s): [3], [4], [6])

[37] Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866. (page(s): [1])

[38] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. (2010). Concurrent abstract predicates. In *ECOOP 2010–Object-Oriented Programming*, pages 504–528. Springer. (page(s):)

[39] Dodds, M., Feng, X., Parkinson, M., and Vafeiadis, V. (2009). Deny-guarantee reasoning. In *Programming languages and systems*, pages 363–377. Springer. (page(s):)

[40] ETCS, O. Open etcs. http://openetcs.org. [Online; accessed 21-Jan-2016]. (page(s):)

[41] ETCS, O. Open etcs: Model change. https://github.com/openETCS/modeling/blob/master/UserStories/DescriptionUserStory14.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[42] ETCS, O. Open etcs: Model change - sequence diagram. https://github.com/openETCS/modeling/blob/master/UserStories/UserStory14.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[43] ETCS, O. Open etcs: Revocation of movement authority. https://github.com/openETCS/modeling/blob/master/UserStories/DescriptionUserStory13.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[44] ETCS, O. Open etcs: Revocation of movement authority - sequence diagram. https://github.com/openETCS/modeling/blob/master/UserStories/UserStory13.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[45] ETCS, O. Open etcs: Route cancellation from end of route signal. https://github.com/openETCS/modeling/blob/master/UserStories/DescriptionUserStory15.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[46] ETCS, O. Open etcs: Route cancellation from end of route signal - sequence diagram. https://github.com/openETCS/modeling/blob/master/UserStories/UserStory15.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[47] Fantechi, A. (2013). Twenty-five years of formal methods and railways: What next? In *Software Engineering and Formal Methods*, pages 167–183. Springer. (page(s): [2], [7])

[48] Ferrari, A., Magnani, G., Grasso, D., and Fantechi, A. (2011). Model checking interlocking control tables. In *FORMS/FORMAT 2010*, pages 107–115. Springer. (page(s):)

[49] Floyd, R. (1967). Assigning meanings to programs. 14:65–81. (page(s):)

[50] Fokkink, W., Groote, J. F., Hollenberg, M., and van Vlijmen, B. (1999). *LARIS 1.0, LAnguage for Railway Interlocking Specifications*, volume 4204 of *Lecture Notes in Computer Science*. Springer Verlag, New York NY. (page(s): [40])

[51] Gay, S. and Hole, M. (2005). Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225. (page(s): [3])

[52] Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N., and Caldeira, A. Z. (2010). Modular session types for distributed object-oriented programming. *SIGPLAN Not.*, 45(1):299–312. (page(s): [3], [6], [40])

[53] Haxthausen, A., Peleska, J., and Pinger, R. (2013). *Applied Bounded Model Checking for Interlocking System Designs*, pages 21–26. Technical University of Denmark. (page(s):)

[54] Haxthausen, A. E. and Peleska, J. (2015). Model checking and model-based testing in the railway domain. In *Formal Modeling and Verification of Cyber-Physical Systems*, pages 82–121. Springer Fachmedien Wiesbaden. (page(s):)

[55] Haxthausen, A. E., Peleska, J., and Kinder, S. (2011). A formal approach for the construction and verification of railway control systems. *Formal aspects of computing*, 23(2):191–219. (page(s):)

[56] Hoare, C. (1969). An axiomatic basis for computer programming. pages 419–438. (page(s):)

[57] Hoare, T. and O'Hearn, P. (2008). Separation logic semantics for communicating processes. *Electronic Notes in Theoretical Computer Science*, 212(0):3 – 25. Proceedings of the First International Conference on Foundations of Informatics, Computing and Software (FICS 2008). (page(s): [7])

[58] Hon, Y. M. and Kollmann, M. (2006). Simulation and verification of uml-based railway interlocking designs. In *Automatic Verification of Critical Systems*, pages 168–172. (page(s):)

[59] Honda, K., Hu, R., Neykova, R., Chen, T.-C., Demangeon, R., Deniélou, P.-M., and Yoshida, N. (2014). Structuring communication with session types. In Agha, G., Igarashi, A., Kobayashi, N., Masuhara, H., Matsuoka, S., Shibayama, E., and Taura, K., editors, *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 105–127. Springer Berlin Heidelberg. (page(s): [6])

[60] Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer. (page(s): [6], [7], [11])

[61] Honda, K. and Yoshida, N. (1995). On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486. (page(s):)

[62] Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., and Honda, K. (2010). Type-safe eventful sessions in java. In Dâ€™Hondt, T., editor, *ECOOP 2010 â€“ Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer Berlin Heidelberg. (page(s): [6])

[63] Hu, R. and Yoshida, N. (2016). Hybrid session verification through endpoint api generation. In *FASE 2016*, volume 9633 of *LNCS*, pages 401–418. Springer. (page(s): [6])

[64] Hu, R., Yoshida, N., and Honda, K. (2008). Session-based distributed programming in java. In *ECOOP 2008–Object-Oriented Programming*, pages 516–541. Springer. (page(s):)

[Intel] Intel. Embedded intel486 processor hardware reference manual. http://www.pld.ttu.ee/~prj/486dev.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[66] James, P., Moller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., Treharne, H., Trumble, M., and Williams, D. (2013). Verification of scheme plans using csp|| b. In *Software Engineering and Formal Methods*, pages 189–204. Springer. (page(s): [7])

[67] Kiss, T. (2015). Formal verification of laris programs using session types and first order logic. In *KEPT*. Babes-Bolyai University. (page(s):)

[68] Kiss, T. (2016). Comparison of session logic with session types. *Studia Informatica*, 61:54–66. (page(s): [39])

[69] Kiss, T., Craciun, F., and Parv, B. (2015a). Extending laris with session types: A case study in the railway interlocking. In *MIPRO*. IEEE. (page(s): [5], [36], [38])

[70] Kiss, T., Craciun, F., and Parv, B. (2015b). Verification of protocol specifications with separation logic. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, pages 109–116. IEEE. (page(s): [4], [37])

[71] Kiss, T. and Jánosi-Rancz, K. T. (2016). Developing railway interlocking systems with session types and event-b. In *IEEE International Symposium on Applied Computational Intelligence and Informatics*, page 163. IEEE. (page(s): [5], [36], [38])

[72] Kobayashi, N., Saito, S., and Sumii, E. (2000). An implicitly-typed deadlock-free process calculus. In *CONCUR 2000—Concurrency Theory*, pages 489–504. Springer. (page(s):)

[73] Kouzapas, D., Yoshida, N., and Honda, K. (2011). On asynchronous session semantics. In *Formal Techniques for Distributed Systems*, pages 228–243. Springer. (page(s):)

[74] Lamport, L. and Schneider, F. B. (1984). The “hoare logic” of csp, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296. (page(s): [7])

[75] Le, D.-K., Chin, W.-N., and Teo, Y.-M. (2012). Variable permissions for concurrency verification. In *Formal Methods and Software Engineering*, pages 5–21. Springer. (page(s):)

[76] Le, D.-K., Chin, W.-N., and Teo, Y.-M. (2013). Verification of static and dynamic barrier synchronization using bounded permissions. In *Formal Methods and Software Engineering*, pages 231–248. Springer Berlin Heidelberg. (page(s):)

[77] Lie, D., Chou, A., Engler, D., and Dill, D. L. (2001). A simple method for extracting models from protocol code. In *2001. Proceedings. 28th Annual International Symposium on Computer Architecture*, pages 192–203. IEEE. (page(s): [5])

[78] Linux, C. Linux driver: aerdrv.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/pci/pcie/aer/aerdrv.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s): [40])

[79] Linux, C. Linux driver: atmel_usba_udc.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/usb/gadget/udc/atmel_usba_udc.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s):)

[80] Linux, C. Linux driver: pcie-designware.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/pci/host/pcie-designware.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s):)

[81] Linux, C. Linux driver: pcie-xilinx.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/pci/host/pcie-xilinx.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s):)

[82] Linux, C. Linux driver: spi-bcm2835.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/spi/spi-bcm2835.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s): [40])

[83] Linux, C. Linux manual page: epoll. http://man7.org/linux/man-pages/man7/epoll.7.html. [Online; accessed 21-Jan-2016]. (page(s): [40])

[84] Linux, C. Linux manual page: poll and ppoll. http://man7.org/linux/man-pages/man2/poll.2.html. [Online; accessed 21-Jan-2016]. (page(s): [40])

[85] Linux, C. Linux manual page: select and pselect. http://manpages.courier-mta.org/htmlman2/select.2.html. [Online; accessed 21-Jan-2016]. (page(s): [40])

[86] López, H. A., Marques, E. R., Martins, F., Ng, N., Santos, C., Vasconcelos, V. T., and Yoshida, N. (2015). Protocol-based verification of message-passing parallel programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 280–298. ACM. (page(s): [6])

[87] Lozes, É. and Villard, J. (2015). Shared contract-obedient channels. *Science of Computer Programming*, 100:28–60. (page(s): [4], [7])

[88] Marques, E. R. B., Martins, F., Vasconcelos, V. T., Ng, N., and Martins, N. D. (2013). Towards deductive verification of mpi programs against session types. In *PLACES'13*, volume 137 of *EPTCS*, pages 103–113. Open Publishing Association. (page(s): [6])

[89] Melham, T. F. (1992). A mechanized theory of the pi-calculus in hol. Technical report, NORDIC JOURNAL OF COMPUTING. (page(s): [7])

[90] Milner, R. (1980). A calculus of communication systems. lncs, vol. 92. (page(s): [3])

[91] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Information and computation*, 100(1):1–40. (page(s):)

[92] Moler, F., Nguyen, H. N., Roggenbach, M., Schneider, S., and Treharne, H. (2012). Combining event-based and state-based modelling for railway verification. (page(s): [7])

[93] Moller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., and Treharne, H. (2012a). Defining and model checking abstractions of complex railway models using csp|| b. In *Hardware and Software: Verification and Testing*, pages 193–208. Springer. (page(s): [7])

[94] Moller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., Treharne, H., Lttgen, G., Merz, S., Margaria, T., Padberg, J., and Taentzer, G. (2012b). Railway modelling in csp|| b: the double junction case study. (page(s): [7])

[95] Mostrous, D. (2005). Moose: a minimal object oriented language with session types. *Master's thesis, Imperial College, London*. (page(s): [7])

[96] Mostrous, D. and Yoshida, N. (2015). Session typing and asynchronous subtyping for the higher-order π-calculus. *Information and Computation*, 241:227–263. (page(s): [3])

[97] Neykova, R. (2013). Session types go dynamic or how to verify your python conversations. In *PLACES'13*, volume 137 of *EPTCS*, pages 95–102. Open Publishing Association. (page(s): [6])

[98] Neykova, R., Yoshida, N., and Hu, R. (2013). Spy: Local verification of global protocols. In Legay, A. and Bensalem, S., editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer Berlin Heidelberg. (page(s): [6])

[99] Ng, N., de Figueiredo Coutinho, J. G., and Yoshida, N. (2015). Protocols by default. In *Compiler Construction*, pages 212–232. Springer. (page(s): [6])

[100] Ng, N., Yoshida, N., and Honda, K. (2012). Multiparty session c: Safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns*, pages 202–218. Springer. (page(s): [6], [7])

[101] O'hearn, P. W. (2004). Resources, concurrency and local reasoning. In *CONCUR 2004-Concurrency Theory*, pages 49–67. Springer. (page(s):)

[102] Palamidessi, C. and Herescu, O. M. (2005). A randomized encoding of the π-calculus with mixed choice. *Theoretical Computer Science*, 335(2–3):373 – 404. Process Algebra. (page(s):)

[103] Palumbo, M. (2014). The ertms/etcs signalling system. *Railway Signalling EU*, 1:1–60. (page(s):)

[104] Pierce, B. C. (1998). Programming in the pi-calculus. *An Experiment in Concurrent Language Design. Tutorial Notes for Pict Version*, 3. (page(s):)

[105] Pucella, R. and Tov, J. A. (2008). Haskell session types with (almost) no class. *SIGPLAN Not.*, 44(2):25–36. (page(s): [3], [6])

[106] Quaglia, P. and Walker, D. (1998). On encoding pπ in mπ. In *Foundations of Software Technology and Theoretical Computer Science*, pages 42–53. Springer. (page(s):)

[107] Sangiorgi, D. (1993). Expressing mobility in process algebras: first-order and higher-order paradigms. (page(s):)

[108] Schneider, S. (2001). *The B-method: An introduction*. Palgrave Oxford. (page(s): [1], [2])

[109] Simpson, A. et al. (1997). The mechanical verification of solid state interlocking geographic data. (page(s): [7])

[110] Takeuchi, K., Honda, K., and Kubo, M. (1994). An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413. Springer. (page(s):)

[111] Turon, A. and Wand, M. (2011). A resource analysis of the π-calculus. *Electronic Notes in Theoretical Computer Science*, 276(0):313 – 334. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII). (page(s): [7])

[112] Vafeiadis, V. and Parkinson, M. (2007). A marriage of rely/guarantee and separation logic. In *CONCUR 2007–Concurrency Theory*, pages 256–271. Springer. (page(s):)

[113] Vallecillo, A., Vasconcelos, V. T., and Ravara, A. (2003). Typing the behavior of objects and components using session types. (page(s): [6])

[114] Vasconcelos, V. T. Bica. http://gloss.di.fc.ul.pt/bica. [Online; accessed 21-Jan-2016]. (page(s): [7])

[115] Vasconcelos, V. T. Mool. http://gloss.di.fc.ul.pt/mool. [Online; accessed 21-Jan-2016]. (page(s): [7])

[116] Vasconcelos, V. T. Partypes. http://gloss.di.fc.ul.pt/ParTypes. [Online; accessed 21-Jan-2016]. (page(s): [7])

[117] Villard, J. (2009a). Heap-Hop Dualized case study. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/send_list_dualized.hop. [Online; accessed 21-Jan-2016]. (page(s): [3])

[118] Villard, J. (2009b). Heap-Hop Home page. http://www.lsv.ens-cachan.fr/Software/heap-hop/index.html. [Online; accessed 21-Jan-2016]. (page(s):)

[119] Villard, J. (2009c). Heap-Hop List Bug Counterexample. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/send_list_bug.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[120] Villard, J. (2009d). Heap-Hop Non Deterministic Counterexample. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/send_list_non_det.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[121] Villard, J. (2009e). Heap-Hop Shared Read of Variable. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/shared_reads.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[122] Villard, J. (2009f). Heap-Hop TCP Example. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/tcp.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[123] Villard, J., Lozes, É., and Calcagno, C. (2009). Proving copyless message passing. In *Programming Languages and Systems*, pages 194–209. Springer. (page(s): [7], [13])

[124] Vu, L. H. (2015). *Formal Development and Verification of Railway Control Systems*. PhD thesis, Technical University of Denmark, 2800 Kongens Lyngby, Denmark. In the context of ERTMS/ETCS Level 2. (page(s): [7])

[125] Vu, L. H., Haxthausen, A. E., and Peleska, J. (2014). Formal modeling and verification of interlocking systems featuring sequential release. In *Formal Techniques for Safety-Critical Systems*, pages 223–238. Springer. (page(s):)

[126] Winter, K. (2002). Model checking railway interlocking systems. In *Australian Computer Science Communications*, volume 24, pages 303–310. Australian Computer Society, Inc. (page(s): [7])

[127] Winter, K. (2012). Optimising ordering strategies for symbolic model checking of railway interlockings. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 246–260. Springer. (page(s):)

[128] Winter, K. and Robinson, N. J. (2003). Modelling large railway interlockings and model checking small ones. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 309–316. Australian Computer Society, Inc. (page(s):)

[129] Zhivich, M. and Cunningham, R. K. (2009). The real cost of software errors. (page(s): [1])